Avalon:
Ein skalierbares Rahmensystem für dynamische Mixed-Reality Anwendungen

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing.)

von
MSc. Johannes Behr
aus Hassfurt in Bayern

Referenten der Arbeit:	Prof. Dr. Marc Alexa
	Prof. Dr. Bernd Fröhlich
	Prof. Dr. José L. Encarnação
Tag der Einreichung:	01. April 2005
Tag der mündlichen Prüfung:	20. Mai 2005

D17
Darmstädter Dissertationen 2005

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Zentrum für Graphische Datenverarbeitung und am Fraunhofer Institut für graphische Datenverarbeitung in Darmstadt. Eine ganze Reihe an Menschen hat an dem Gelingen dieser Arbeit Anteil. Bei diesen Menschen möchte ich mich für ihre Unterstützung bedanken. Vor allem bei meiner Familie und meiner Freundin, die mich über den langen Zeitraum immer wieder ermutigten und bestärkten.

Prof. Dr. Marc Alexa danke ich für die vielen Diskussionen, hilfreichen Ideen und Anregungen und seine Bereitschaft in jeder Situation hilfreich zur Seite zu stehen. Seine oft auch kritischen Anmerkungen haben mich immer wieder angespornt und motiviert, an der Thematik weiterzuarbeiten. Dass ich in den letzten sieben Jahren an interessanten und immer wieder spannenden Aufgaben arbeiten durfte, möchte ich mich bei Prof. Dr. José L. Encarnação bedanken. Er hat es mir ermöglicht, dass ich in einem erstklassigen Umfeld mit einem hoch motivierten Team arbeiten und forschen konnte. Meinen Kollegen in der Abteilung ”Visual Computing” und ”Visualisierung und Virualle Realität danke ich für die grandiose Zusammenarbeit und die freundschaftliche Atmosphäre.

Mein besonderer Dank gilt meinen Kollegen Dirk Reiners, Gerett Voss und Patrik Dähne. Der wissenschaftliche Austausch mit ihnen war für mich außerordentlich wertvoll und bereichernd.

Bedanken möchte ich mich auch bei den vielen Studenten, die ich in den letzten Jahren als wissenschaftliche Hilfskräfte, Praktikanten und Diplomanten betreuen konnte. Ohne ihren Fleiß und Zuverlässigkeit wäre es nicht möglich gewesen, die vielen Ideen umzusetzen und in Projektarbeiten einzubringen. Ihre Beiträge und Anregungen finden sich auch in der vorliegenden Arbeit wieder.

Inhaltsverzeichnis

1 Einleitung **10**

1.1 Problemstellung und Motivation 10

1.2 Ziel und Aufbau der Arbeit 13

1.3 Zusammenfassung der wichtigsten Ergebnisse 14

2 Stand der Technik **17**

2.1 Grundlagen 17

2.2 Übersicht über existierende dynamische VR/MR-Systeme 19

2.2.1 Alice 19

2.2.2 Avacado / Avango 22

2.2.3 dVS/dVISE 24

2.2.4 Lightning 25

2.2.5 Maverik 27

2.2.6 VR-Juggler 31

2.2.7 Unreal Engine 32

2.2.8 Flux und X3D-Systeme 35

2.3 Zusammenfassung 37

3 Effiziente Modellierung von dynamischen Anwendungen **40**

3.1 Grundlagen 40

3.2 Mehrdimensionale Graphen zur Modellierung der Dynamik 42

3.3 Komponentenausprägungen und Erweiterbarkeit 45

3.3.1 Komponentenausprägung, Identität und Status 45

3.3.2 Komponentenschnittstellen, Felder und Slots 47

3.3.3 Statische und dynamische Felder 49

3.3.4 Erweiterbarkeit 49

3.4 Deklaration und Implementierung von Komponententypen 51

3.4.1 Erweiterung der Basisknoten 51
3.4.2 Implementierung durch Scrnipt-Sprachen 52
3.4.3 Aggregation 53
3.5 Namensräume und Laufzeit-Kontext 54
3.5.1 Hierarchie von typisierten Namensräumen 54
3.5.1.1 Szene 55
3.5.1.2 Engine 56
3.5.1.3 Prototypen-Deklarationen und Instanzen 57
3.5.2 Laufzeit-Kontext 57
3.6 Typisierte Kanten und Nachrichtenverarbeitung 58
3.6.1 Typisierte Kanten 58
3.6.2 Nachrichtenverarbeitung und Synchronisation 58
3.7 Traversierung der Graphen 59
3.8 Netzwerktransparente Schnittstellen 60
3.8.1 HTTP-Service 62
3.8.2 SOAP Service 64
3.8.3 External Scene Interface Service 65
3.9 Laufzeitumgebungen 65
3.10 Zusammenfassung 67

4 Flexible Interaktion und Sensorik 69
4.1 Grundlagen 71
4.1.1 Interaktionsgeräte 72
4.1.1.1 Eingabegeräte 72
4.1.1.1.1 Geräte für Schreibtischumgebungen . 72
4.1.1.1.2 Positions- und Orientierungsmesssysteme 74
4.1.1.1.3 3D-Mäuse 77
4.1.1.1.4 Medienströme 78
4.1.1.2 Ausgabegeräte 78
4.1.1.2.1 Haptische Ausgabegeräte 79
4.1.1.2.2 Kraftrückkopplung 79
4.1.1.2.3 Taktile Rückkopplung 80
4.1.2 System zur Geräteabstraktion 80
4.2 Sensoren für Datenströme 81
4.3 Sensoren zur direkten Manipulation 86

4.3.1 Sensoren zur Selektion eines Teilgraphen 86
4.3.1.1 2D-Interaktion 87
4.3.1.2 Immersive Interaktion 88
4.3.1.2.1 Selektion auf Distanz 88
4.3.1.2.2 Selektion im Nahbereich 90
4.3.2 Sensoren zur Transformation eines Teilgraphen 90
4.3.2.1 2D-Interaktion 91
4.3.2.2 Immersive Interaktion 91
4.3.2.2.1 Transformation auf Distanz 91
4.3.2.2.2 Transformation im Nahbereich 92
4.4 Sensoren zur indirekten Manipulation 93
4.4.1 Positionierung 97
4.4.2 Selektion 97
4.4.3 Quantifizierung 98
4.4.4 Text 98
4.4.5 Rotation 98
4.5 Zusammenfassung 98

5 Parallelverarbeitung und Skalierbarkeit 100
5.1 Grundlagen 100
5.1.1 Rechenleistung 101
5.1.2 Graphikleistung 101
.0.3 Gesamtarchitekturen 102
.1 Parallelverarbeitung 103
.1.1 Parallelverarbeitung auf dem Applikationsgraphen 104
.1.2 Parallelverarbeitung der Ausgabe 105
.2 Skalierung der Applikations- und Darstellungsleistung 106
.3 Skalierbare Verfahren zur Darstellung 108
.3.1 Progressive Darstellung von Polygonnetzen 109
.3.1.1 Grundlagen 109
.3.1.1.1 Polygon- und Dreiecksnetze 109
.3.1.1.2 Progressive Netz-Repräsentation . . . 111
.3.1.1.3 Netz Simplifizierung 112
.3.1.1.4 Explizite Energiefunktionen 113
.3.1.1.5 Fehlerquadriken 113
.3.1.1.6 Lindstrom und Turk 115

.3.1.1.7 Melax 117
.3.1.2 Kodierung von Progressiven Netzen 117
.3.1.3 Integration 120
.3.1.4 Skalierbarkeit und Ergebnisse 122
.3.2 Volumenvisualisierung 123
.3.2.1 Grundlagen 123
.3.2.1.1 Volumendaten 124
.3.2.1.2 Verfahren zur Volumenvisualisierung . 125
.3.2.1.3 Physikalische Grundlagen 127
.3.2.1.4 Prä- und Post-Klassifikation 128
.3.2.2 Texturbasierte Verfahren 130
.3.2.3 Integration 133
.3.2.3.1 1D/2D/3D Textur-Knoten 133
.3.2.3.2 SliceSet Knoten 134
.3.2.4 Skalierbarkeit und Ergebnisse 136
.4 Zusammenfassung 138

A Applikationsbeispiele **139**
A.1 Dom von Siena 139
A.2 Archeoguide 141
A.3 Embassi-Fahrsimulator 142
A.4 Zusammenfassung 143

B Zusammenfassung und Ausblicke **144**
B.1 Zusammenfassung 144
B.2 Ausblicke 146
B.2.1 Integration von physikalischen Beziehungen 146
B.2.2 Parallelverarbeitung von zyklischen Graphen 147
B.2.3 Integration von weiteren skalierbaren Teilverfahren 147

Literaturverzeichnis **148**

Abbildungsverzeichnis

2.1 Alice Systemarchitektur 20
2.2 Szene organisiert als Graph in Alice 22
2.3 Avango-FieldContainer als Ableitung der Perforce-Klasse 23
2.4 Lightning Systemarchitektur 26
2.5 Objektpool zur Organisation der Szeneelemente in Lightning . . . 27
2.6 Das Propagieren von Nachrichten in Lightning 28
2.7 Maverik Systemarchitektur 28
2.8 Beispielanwendung in Maverik 30
2.9 VR-Juggler Systemarchitektur 31
2.10 Statische Unreal Szenendarstellung 32
2.11 Unreal Szene mit Interaktiven Element 33
2.12 X3D-Anwendung im Flux-Player 35

3.1 MR-Systemdesign mit applikationsspezifischer Kontrolleinheit . . 41
3.2 Kommunikationskanäle zwischen Knoten 42
3.3 Systemdesign mit applikationsunabhängiger Kontrolleinheit . . . 44
3.4 Zustände einer Komponente 46
3.5 Strukturen zur Verwaltung der reflektiven Schnittstellen von Komponenten 50
3.6 Graphisch-interaktives Werkzeug zu Erstellung der reflektiven Schnittstellen 52
3.7 Deklaration von Komponenten durch Aggregation 53
3.8 Gülige Hierarchie von Namensräumen 55
3.9 Application und System-Services 61
3.10 Dynamische Liste aller Verfügbaren *Service*-Schnittstellen 61
3.11 Graphisch-Interaktive HTTP Schnittstelle eines Knoten 63
3.12 Graphisch-Interaktive Laufzeitumgebung zum Editieren und Überwachen von Anwendungen 66

4.1 Spacemouse Interaktionsgerät 74
4.2 Elektromagnetische Messsysteme eingesetzt in einer VR-Simulation 75
4.3 Kabelloses Interaktionsgerät mit optischen Markern 76
4.4 Datenhandschuh zur Messung von Fingerstellungen 77
4.5 SFImageSensor Anwendung in einer AR-Umgebung 85
4.6 Fahrsimulator als Anwendung von DSS Objekten 85
4.7 Vererbungshierachie der X3D-PointingSensor-Knoten 86
4.8 2D-Mouse Selektion 88
4.9 Strahlbestimmung zur immersiven Interaktion mit einem PointingSensor 89
4.10 Interaktion mit einem TouchSensor auf Distanz 89
4.11 Kollision und Aktivierung des Sensors im Nahbereich 90
4.12 Projektionsfläche zur Bestimmung der Sensorrotation 92
4.13 SphereSensor Interaktion im Nahbereit 93
4.14 IMS Sensor Repräsentation für Desktop- und Immersive-Umgebungen 95
4.15 2D-Repräsentation einer Quantifizierung 96
4.16 3D-Repräsentation einer Quantifizierung 97

1 Entwicklung der Graphikleistung zwischen 1996 und 2004 102
2 Spektrum der MR-Hardwareplattformen 103
3 Automatische Parallelverarbeitung auf den Event-Pfaden 105
4 Applikations- und Darstellungsgraph 106
5 Sequentielle und parallele Verarbeitung von Zyklen 107
6 Applikations- und darstellungslimitierte Zyklen 107
7 Scharfe Kanten durch unterschiedliche skalare Attribute 111
8 ecol- und vsplit Transformation 112
9 Durch drei Optimierungsebene definierte Schnittpunkt 116
10 TriangleStrip Pyramide zur Optimierung der Vertex-Cache Auslastung 119
11 Progressives Netz mit 1000, 30000, 65451 Dreiecken 122
12 Darstellungszeiten für ein Progressives Netz mit Dreiecken und Triangle-Strip Primitiven 123
13 Strukturierte Volumengitter 124
14 Unstrukturierte Volumengitter 125
15 Parameter des Volumen-Rendering-Integrals 128

16 Prä- und Post-Klassifikation 129
17 View-alligned Schnitte für 2D-Texturen 132
18 Viewport alligned Polygone für 3D-Texturen 132
19 Anwendung von unterschiedlichen Blend-Funktionen zur Volumenvisualisierung 133
20 Lineare Abhängigkeit der Darstellungszeiten für Volumen 137

A.1 Anwendung der Gesichts- und Körperanimationstechniken 140
A.2 Archeoguide; eine Freiland-AR Anwendung 141
A.3 Embassie Fahrsimulator als Anwendung für hoch dynamische Systeme 142

Tabellenverzeichnis

2.1 Klassifikation der Eigenschaften der untersuchten VR-Systeme . . 38

3.1 Vom System bereitgestellte Felddatentypen 48

4.1 Sensortypen für direkte Datenströme 84

Kapitel 1

Einleitung

1.1 Problemstellung und Motivation

Das Forschungsgebiet Virtuelle Realität (*Virtual Reality*, *VR*, [121][26]) hat durch seine spektakulären Entwicklungen viele Menschen nachhaltig beeindruckt. Head-Mounted-Displays und Multi-Screen-Projektionssysteme beflügelten die Phantasie von Anwendern und Entwicklern und brachten eine Vielzahl neuer Ideen, neuer Hardware und neuen Anwendungen hervor.

Erweiterte Realität (*Augmented Reality*, *AR*, [96]) stellt eine verwandte, jedoch jüngere Technologie dar, in der die Anwendung die Realität durch virtuelle Elemente erweitert. Im Unterschied zu VR-Anwendungen agiert der Benutzer nicht in einer vollständig synthetischen Welt, sondern es erfolgt eine Zusammenführung der Realität mit computergenerierten, virtuellen Objekten.

Seit einigen Jahren gibt es Bemühungen, beide Forschungsgebiete unter dem Begriff Mixed Reality (*MR*, [92]) zusammenzuführen. Der Anteil und die Gewichtung von virtuellen und realen Objekten ist in MR-Applikationen variabel und kann von *ausschließlich virtuell* (VR) bis *überwiegend real* (AR) variieren.

MR- und AR-Anwendungen stellen neue Anforderungen an Interaktionsgeräte und Meßsysteme. Sie basieren jedoch technologisch, was zum Beispiel die Darstellung und Verwaltung von 3D-Szenen anbelangt, auf Ergebnissen der VR-Forschung.

MR als Technik hat ein klares Ziel: Die Verbesserung der Mensch-Maschine-Schnittstelle. VR- und AR-Umgebungen versprechen insbesondere wenn es darum geht ausgesprochen komplexe, technische und wissenschaftliche Sachverhalte zu verstehen oder zu bearbeiten, eine deutliche Verbesserung. Immersive Systeme er-

lauben dabei dem Benutzer, sich mit Informationen zu umgeben und in diese einzutauchen. Dies stellt für viele Anwendungen eine Verbesserung gegenüber traditionell zweidimensionalen Benutzerschnittstellen dar. Idealerweise wird der Benutzer dabei visuell, akustisch und taktil in einen virtuellen Datenraum eingebunden. Er kann die erzeugten virtuellen Objekte im dreidimensionalen Raum begehen, erleben und bearbeiten, um zum Beispiel virtuelle Prototypen von Produkten zu gestalten, zu visualisieren und zu testen.

Zur Umsetzung der VR-Technik wurden seit Anfang der neunziger Jahre eine Vielzahl von unterschiedlichen VR-Softwaresystemen entwickelt, die bis heute vorwiegend im industriellen Umfeld, in Deutschland schwerpunktmäßig in der Automobilindustrie, zum Einsatz kommen. Die Automobilindustrie hat in den vergangenen Jahren Millionen Euro in VR-Labors investiert, um den Einsatz von VR-Technologien zu erproben und geeignete Lösungen zu entwickeln.

Dabei sind Anwendungen entstanden, die einen messbaren *return of invest* für die einzelnen Firmen erwirtschaften. Dennoch hat sich bis heute kein Massenmarkt für VR-Technologien und Systeme entwickelt, obwohl einige Untersuchungen [121][26] zeigen, dass für ausgesuchte Aufgaben das Arbeiten in immersiven Umgebungen einen messbaren Vorteil gegenüber klassischen Desktop-Umgebungen bietet.

Ein Grund sind sicherlich die immensen Investitionskosten, die ein Unternehmen für die Entwicklung von spezifischen VR-Lösungen aufbringen muss. Vor allem Erstinstallations- und Betriebskosten in Millionenhöhe haben anfangs viele Firmen abgeschreckt. In der Zwischenzeit sind jedoch die Preise für Projektionssysteme und insbesondere für Graphikmaschinen soweit gesunken, dass typische VR-Installationen für einen Bruchteil der früheren Investitionen zu beschaffen sind.

Geblieben ist die aufwendige Softwareentwicklung. Die Anforderungen an MR-Softwaresysteme sind über die letzten zehn Jahre gestiegen. Eine grundsätzliche Anforderung ist die Unterstützung einer wachsenden Anzahl von Ein- und Ausgabekanälen deren Ziel es ist, die Umgebungen immer realistischer erscheinen zu lassen. Ausschlaggebend hierfür ist nicht nur die Implementierung der vielfältigen Geräteschnittstellen, sondern besonders die logische Kombination und Koordination von unterschiedlichen Eingabe- und Ausgabekanälen (z.B. Sprache, Gesten, 3D-Sound, 3D-Graphik, etc.). Darüber hinaus sind neuartige Applikationen angereichert mit einer Vielzahl von interaktiven und dynamischen Elementen. Diese hoch interaktiven und dynamischen Systeme müssen den zeitlichen Veränderungen im Allgemeinen und im Speziellen den variierenden Bildraten und

deren Synchronisation standhalten. MR-Systeme müssen in Echtzeit operieren. Dies erfordert, dass die Reaktionszeiten klein genug sind, um die Grenzen zwischen der realen und der virtuellen Welt verschwinden zu lassen.

Über die Jahre wurden unterschiedliche Systeme vorgestellt mit dem Schwerpunkt, die VR-Softwareentwicklung zu vereinfachen. So gibt es heute eine Vielzahl von Systemen zur Echtzeitvisualisierung (z.B. OpenSG [105], Performer [111]) und zur Verwaltung von Eingabegeräten und Kanälen (z.B. OpenTracker [107], VRPN [75]). Dennoch ist die Entwicklung von interaktiven und dynamischen AR- und VR-Applikationen weiterhin aufwendig, zeitraubend und im Allgemeinen nur von erfahrenen Entwicklern mit Spezialkenntnissen zu leisten, da viele derzeitige VR-Systeme (z.B. VRJuggler [17], Maverik [58]) keine einheitliche und umfassende Abstraktion für Verhaltensbeschreibungen beinhalten.

Relevante Basisprobleme der Visualisierung und Interaktion sind zum großen Teil verstanden und entsprechende Lösungen entwickelt. Alle aktuellen Systeme beinhalten Techniken zur optimierten Darstellung von komplexen Szenen in Echtzeit und erlauben es, unterschiedliche Ein- und Ausgabegeräte anzusprechen. Jedoch wird die Koordination der Veränderungen der Szenenbeschreibung und somit die eigentliche Applikationslogik immer noch an die Applikationsentwickler übertragen.

Das bedeutet, dass der Applikationsentwickler den Datenfluss von externen Modulen (z.B. Simulatoren), zeitabhängige Veränderungen (z.B. Animationen) und Datenströme der Ein- und Ausgabegeräte mit Hilfe einer Programmiersprache koordinieren, beschreiben und überwachen muss. Somit sind die Applikationsschnittstellen der meisten gegenwärtigen VR-Systeme von Softwareentwicklern für Softwareentwickler entworfen und modelliert.

Um eine größere Verbreitung und Akzeptanz von MR zu erreichen, ist es notwendig, neuartige Systeme zu entwickeln. Es sind einheitliche und umfassende Modelle zur Verhaltens- und Interaktionsmodellierung erforderlich, die es dem Applikationsentwickler erlauben, anwendungsspezifische Anforderungen effizient und flexibel umzusetzen. Entsprechende Rahmensysteme übertragen die Verantwortung der Koordination und der Ablaufsteuerung vom Applikationsentwickler auf die Laufzeitumgebung. Es ist wünschenswert, dass diese neuen Systeme skalierbare Methoden bereitstellen, um lokale und globale Anwendungsziele, zum Beispiel eine konstante Bildwiederholrate, erfüllen zu können.

1.2 Ziel und Aufbau der Arbeit

Ziel dieser Arbeit ist es, die technologischen Anforderungen für ein flexibles und modernes MR-System zu definieren, und neuartige Lösungen für die daraus abgeleiteten Probleme zu erarbeiten.

Kapitel 2 präsentiert eine Übersicht der verfügbaren VR-Systeme und Modelle. Dabei werden unterschiedliche VR-Systeme vor allem daraufhin untersucht, inwieweit sie Abstraktionen und Methoden zur Modellierung dynamischer Veränderungen bereitstellen. Weiterhin werden Entwicklungen und Probleme analysiert und Ziele für weitere Entwicklungen definiert. Drei Problemklassen werden dabei isoliert und deren Lösung als Voraussetzung für zukünftige Systeme identifiziert: Modellierung von dynamischen Verhalten, eine darauf angepasste effiziente Sensorik und die Skalierbarkeit des Gesamtsystems. Diese Bereiche werden in den folgenden Kapiteln genauer untersucht und Lösungen präsentiert.

Kapitel 3 analysiert und vergleicht die unterschiedlichen Ansätze zur Beschreibung von Verhalten sowie die Programmiermöglichkeiten, die von VR-Systemen bereitgestellt werden. Dabei werden die vorherrschenden Systemarchitekturen und Programmiermodelle untersucht und deren Probleme erläutert. Darauf aufbauend wird eine neuartige Systemarchitektur mit einem einheitlichen Komponenten- und Kommunikationsmodell entwickelt und deren Elemente beschrieben. Die neue Architektur fordert Strukturen und Kommunikationsmechanismen, die daraufhin genauer untersucht werden. Anschließend werden Detaillösungen für die unterschiedlichen Fragestellungen entwickelt und vorgestellt.

Kapitel 4 stellt zum einen unterschiedliche physikalische Geräteklassen und deren Merkmale für Ein- und Ausgaben vor, zum anderen werden bestehende Systeme und Abstraktionen für virtuelle Geräteklassen untersucht und klassifiziert. Aufbauend auf diesen Untersuchungen und unter Einbeziehung der Ergebnisse aus Kapitel 3 wird ein neuartiges dreistufiges Sensorkonzept entwickelt. Anschließend werden die drei Stufen explizit vorgestellt und deren Integration und Anwendung an Beispielen erläutert.

Kapitel 5 präsentiert eine Zusammenfassung der aktuellen Prozessor- und Grafikhardwareentwicklung und leitet daraus einen stetig wachsenden Bedarf

für die Parallelisierung der Softwaresysteme ab. Es werden die unterschiedlichen Aufgaben innerhalb eines typischen MR-Systems analysiert und auf ihre Parallelisierbarkeit untersucht. Anschließend wird ein neuartiges Modell entwickelt, das es erlaubt, die Kosten der unterschiedlichen Prozesse des System zu skalieren. Erst dadurch kann die Erfüllung der globalen Zielvorgaben und die optimale Nutzung der bereitgestellten Resourcen erreicht werden. Um die Anforderung der globalen Skalierbarkeit umsetzen zu können, muss das System lokale, auf die Darstellung oder Modellveränderung spezialisierte Verfahren einsetzen. Zu diesem Zweck werden existierende Methoden vorgestellt, untersucht und entscheidende Verbesserungen entwickelt. Anschließend wird deren Integration diskutiert und konkrete Lösungen präsentiert.

Kapitel 6 beschreibt ausgewählte Anwendungen des Rahmensystems und deren Anforderungen, Design und Implementierung. Dabei werden möglichst unterschiedliche Aspekte aus dem MR-Spektrum berücksichtigt.

Kapitel 7 fasst die Ergebnisse der Arbeit aus den Bereichen der dynamischen Modellbeschreibung, der Sensorik und der Skalierbarkeit zusammen und zeigt Forschungsfelder für weitere und anschließende Arbeiten auf.

1.3 Zusammenfassung der wichtigsten Ergebnisse

Im Rahmen dieser Arbeit wird die Thematik der Beschreibung und Modellierung von dynamischen und interaktiven MR-Applikationen ganzheitlich betrachtet, sowie Lösungen erarbeitet und evaluiert. Im Einzelnen wurden die folgenden Ergebnisse erzielt:

Entwicklung eines einheitlichen Modells zur Beschreibung von Struktur und Verhalten für MR-Anwendungen

Es wird gezeigt, dass traditionelle MR-Systeme vorwiegend Szenengraphen zur Organisation von strukturellen und graphischen Elementen einsetzen. Zusätzlich bieten Sie eine Klassifikation von virtuellen Geräten und deren Abstraktion an, um physikalische Ein- und Ausgabegeräte anzusprechen. Es wird weiterhin gezeigt, dass bisher zur Modellierung der verbindenden Applikationslogik und -dynamik nur ungenügende Modelle bereitstehen.

Aus diesem Grund wird ein einheitliches Modell von Graphen entwickelt, das es erlaubt, alle dynamischen Aspekte einer MR-Applikation mit Hilfe von Komponenten und typisierten Kanten zu modellieren. Das Konzept abstrahiert die klassischen topologischen Beziehungen sowie statische und dynamische Nachrichtenkanäle als Kanten innerhalb unterschiedlicher gerichteter Graphen. Daraus abgeleitete Anforderungen werden diskutiert und entsprechende Lösungen entwickelt. Dabei werden insbesondere Problemstellungen wie Erweiterbarkeit, die Verwaltung und Bereitstellung von statischen und dynamische Metabeschreibungen und die korrekte Verarbeitung von Nachrichten untersucht und neuartige Ansätze entwickelt.

Durch das einheitliche Modell wird der Entwicklungsprozess zur Erstellung von MR-Anwendungen gegenüber dem Stand der Technik wesentlich vereinfacht.

Entwicklung eines mehrstufigen Sensor-Modells zur flexiblen Interaktion mit MR-Welten

Es wird gezeigt, dass bisherige MR-Systeme sich meist auf eine statische Abstraktion von virtuellen Gerätegruppen beschränken und kaum darauf aufbauende Interaktionsmodelle anbieten. Aus diesem Grund wird in dieser Arbeit ein neuartiges, mehrstufiges Sensorkonzept entwickelt und vorgestellt, das unabhängig von Geräteklassifikationen Interaktionsverfahren beinhaltet, welche abhängig von der Ausprägung der Laufzeitumgebung interaktive Elemente bereitstellt. Dazu werden drei aufeinander aufbauende Sensorgruppen entwickelt, die als Komponenten der dynamischen und hierarchischen Graphen zum Einsatz kommen.

Data Stream Sensor (DSS) Objekte erlauben die direkte Anbindung von Ein- und Ausgabeströmen, unabhängig von einzelnen Geräten oder Interaktionsaufgaben. Sie sind nicht an Geräteklassen gebunden, sondern abstrahieren einzelne, typisierte Datenströme von oder zu Geräten.

Direct Manipulation Sensor (DMS) Objekte erlauben dem Anwender Teile der Szene graphisch-interaktiv und direkt zu manipulieren. Diese Sensoren reagieren auf Veränderungen des Benutzermodells, sind selbst aber unabhängig von konkreten Eingabegeräten.

Indirect Manipulation Sensor (*IMS*) Objekte ermöglichen der Anwendung Parameter vom Benutzer zu erfragen, und definieren somit konkrete Eingabeaufforderungen und Aufgaben. Sie beinhalten aber keine einzelnen graphischen Repräsentationen, sondern wählen selbständig und abhängig von der Laufzeitumgebung eine geeignete Interaktionsform aus.

Das im Rahmen dieser Arbeit entwickelte Modell stellt eine wesentliche Verbesserung gegenüber bestehenden Systemen dar. Es erlaubt die effiziente Entwicklung von Applikationen, die unabhängig von physikalischen und virtuellen Geräten und der eingesetzten Laufzeitumgebung sind.

Entwicklung von parallelen Modellen zur Skalierung der Applikationskosten

Anhand von aktuellen Entwicklungen in der Prozessortechnologie wird gezeigt, dass es für die optimale Auslastung der Resourcen notwendig ist, möglichst viele Aufgaben und Verfahren zur Laufzeit auf unterschiedliche Prozesse aufzuteilen. Dazu werden die typischen Abläufe in einem MR-System untersucht und eine Lösung entwickelt, die zwei unterschiedliche Ansätze der Parallelisierung beinhaltet. Zum einen wird die klassische Trennung von Applikation und Darstellung unterstützt und verallgemeinert, zum anderen wird ein neuartiges Verfahren entwickelt und vorgestellt, das die automatische Parallelisierung auf den Ereigniskanten der Applikationsgraphen bereitstellt. Der entscheidende Vorteil gegenüber den bestehenden Verfahren ist die selbständige und autarke Ermittlung von Teilgraphen, die parallel abgearbeitet werden.

Mithilfe dieser Lösung können Prozesse innerhalb einer Applikation sehr effizient verteilt werden, aber dennoch sind die Resourcen realer Maschinen begrenzt. Um globale Applikationsvorgaben erfüllen zu können, wie zum Beispiel das Darstellen mit einer fixen Bildwiederholrate oder die Animation einer variierenden Anzahl von Figuren pro Zyklus, ist eine globale Skalierbarkeit der Kosten gefordert. Zur Lösung dieses Problems wurde ein globales Modell entwickelt und eingeführt, das automatisch lokale Kosten der Prozesse zur Darstellung und Veränderung skaliert. Um diese Voraussetzung erfüllen zu können, wurden unterschiedliche skalierbare Verfahren untersucht und teilweise entscheidend verbessert.

Das in dieser Arbeit entwickelte Modell zur Parallelisierung und Skalierung von MR-Systemen ist eine wesentliche Verbesserung gegenüber den bisherigen Verfahren, da es auf der Grundlage der globalen Zielvorgaben und dem Applikationsgraphen selbständig und automatisch den Resourcenverbrauch regelt und verteilt. Der Prozess der Anwendungsentwicklung wird wesentlich vereinfacht, da eine aufwendige und fehlerträchtige manuelle Parallelisierung entfällt.

Kapitel 2

Stand der Technik

Dieses Kapitel gibt einen Überblick über bestehende Virtual-Reality-Systeme. Schwerpunkt der folgenden Untersuchung sind die von VR-Systemen eingesetzten dynamischen Elemente, sowie die dem Benutzer zur Verfügung gestellten Möglichkeiten der Applikationsentwicklung. Da die meisten Systeme spezifische Modellbeschreibungen beinhalten, gibt es in der Regel keine Standardwerkzeuge und -schnittstellen, mit denen Applikationen erstellt werden. Aus diesem Grund bieten viele Systeme eigene Abstraktionen an, mit denen Entwickler 3D-Anwendungen erstellen können. Diese Modelle werden untersucht und vorgestellt.

Abschließend erfolgt ein kurzer Überblick über die Systemeigenschaften und deren Klassifikation. Aus dieser Untersuchung werden neuartige Anforderungen entwickelt, die als Grundlage für die Untersuchungen und Entwicklungen in den weiteren Kapiteln dienen.

2.1 Grundlagen

Alle VR-Systeme haben ein grundlegendes Prinzip gemeinsam: um den Zustand der virtuellen Welt zu ändern, werden immer wieder Daten von Eingabegeräten oder externen Schnittstellen gelesen und Animations- bzw. Simulationsbeschreibungen ausgewertet. Die aktuell darzustellenden Zustände werden aus den Daten errechnet und schließlich synthetische Bilder berechnet und ausgegeben. Anschließend beginnt das System den Prozess von Neuem.

Als *Bildrate* oder *Framerate* bezeichnet man die Anzahl der pro Sekunde dargestellten Bilder. Sie ist ein wichtiges Element im Erzeugen von realistisch wirkenden Applikationen, da sie direkten Einfluss auf das vom Benutzer wahrgenommene

Erlebnis hat. Das menschliche Auge kann ab ca. 25 Bildern pro Sekunde das einzelne Bild nicht mehr als solches wahrnehmen. Bei weniger als 15 Bildern nimmt der Benutzer das Einzelbild wahr - die Interaktivität der Anwendung wird erheblich beeinträchtigt.

Die Modellierung der Dynamik beschäftigt sich mit der Veränderung, die zwischen zwei Bildern stattfindet. Allgemein bedeutet Dynamik bzw. dynamische Modellveränderung, dass durch innere oder äußere Einflüsse die zu einem bestimmten Zeitpunkt vorhandenen Daten und Objekte und deren Beziehung untereinander verändert werden, wie zum Beispiel das Verändern der Position, der Orientierung oder das Veränderen eines anderen Attributs eines Objekts, eines Lichts oder der Kamera.

Dynamische Modellveränderungen äußern sich durch eine Veränderung der beschreibenden Daten. Das bedeutet, dass Aktionen definiert werden müssen, die auf die Objekte angewendet werden.

Bei der Dynamische Modellveränderungen durch innere Ereignisse kommen die Änderungen aus der Applikation selbst. Dabei lösen bestimmte Ereignisse, die sich zur Laufzeit der Anwendung ereignen, änderungen in den Applikationsdaten aus. Ein Beispiel dafür sind Ereignisse oder Scipts, bei denen zu einem definierten Zeitpunkt eine Aktion ausgeführt wird.

Bei Veränderungen durch äußere Ereignisse ist der Benutzer verantwortlich für die Veränderung. Er manipuliert zum Beispiel mit den ihm zur Verfügung stehenden Eingabegeräten die Sicht auf die dargestellte Szene, oder er manipuliert die Szene selbst, indem er Objekte selektiert und verändert, ihnen neue Positionen zuweist, neue Objekte erzeugt oder bestehende Objekte löscht.

All diese Aktionen und Ereignisse führen zu Veränderungen in den die Szene beschreibenden Daten. Die Aufgabe eines VR-Systems ist es, diese Modellveränderungen möglichst effizient durchzuführen und die Applikationsentwickler bei der Erstellung von Anwendungen, die Gebrauch von dynamischen Elementen in der VR machen, zu unterstützen. Diese Unterstützung kann sowohl durch leistungsfähige Schnittstellen erfolgen, als auch durch die Bereitstellung von intuitiv zu bedienenden Werkzeugen und der Zusammenarbeit mit auf dem Markt etablierten Standardtools.

2.2 Übersicht über existierende dynamische VR/MR-Systeme

In den letzten Jahren wurde eine Vielzahl kommerzieller und freier VR-Systeme entwickelt. Auch wenn sie ähnliche Zielsetzungen haben, unterscheiden sie sich in ihrer Applikationslogik deutlich voneinander.

Ein Hauptunterscheidungsmerkmal sind sicherlich die verwendeten und unterstützten Umgebungen. Sie reichen von Low-End-Systemen, wie zum Beispiel PDA- oder Desktop-Systemen mit einfachen Bildschirmen, die typischerweise in Privathaushalten zu finden sind, bis zu High-End-Umgebungen wie die Workbench [73] oder CAVE [30], die von speziellen leistungsfähigen Grafikcomputern gesteuert werden und dementsprechend kostspielig sind.

Im Folgenden sollen exemplarisch verschiedene VR-Systeme vorgestellt werden. Die Auswahl erhebt keinen Anspruch auf Vollständigkeit. Vielmehr sollen möglichst unterschiedliche Systeme untersucht werden, die sich nicht nur in der unterstützten Umgebung unterscheiden, sondern auch in der Umsetzung von dynamischen Szenarien. Die betrachteten Systeme reichen dabei von High-End-Systemen wie zum Beispiel Avango oder Alice, die in Forschungsanstalten auf dem aktuellen Stand der Rechnertechnik mit den derzeit vorhandenen Ein- und Ausgabegeräten zum Einsatz kommen, über kommerzielle Systeme wie zum Beispiel dVise und Marveric, die Produktsimulation und -entwicklung in den Vordergrund stellen, bis hin zu einer 3D-Spiele-Engine, Unreal Engine, die auf handelsüblichen Rechnern läuft. Die Systeme benutzen verschiedene Techniken, um Veränderungen an der aktuell dargestellten Szene und der beschreibenden Datenstrukturen vorzunehmen und bieten unterschiedliche Abstraktionsmodelle für den Applikationsentwickler, diese Möglichkeiten auch zu nutzen. Auf den unterschiedlichen Abstraktionsmodellen und ihren Ausprägungen liegt der Hauptaugenmerk in den nachfolgenden Untersuchungen.

2.2.1 Alice

Alice [28][27] ist ein Rapid-Prototyping System zum Erstellen von interaktiven virtuellen Umgebungen, das an der Universität von Virginia entwickelt wurde. Das Ziel von Alice ist es, dem Benutzer ein Werkzeug zur Verfügung zu stellen, mit dem er schnell dynamische Szenen kreieren kann, ohne detaillierte Programmierkenntnisse zu benötigen. Alice basiert auf der objektorientierten, interpretier-

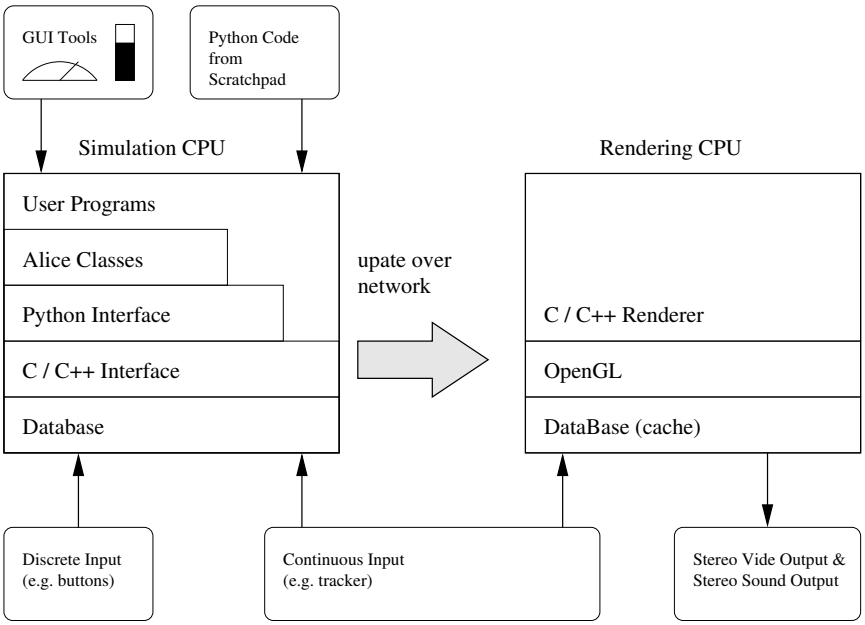

Abbildung 2.1: Alice Systemarchitektur

ten Programmiersprache Python, die um systemspezifische Objekte und Funktionen erweitert wurde. Durch die Trennung von Simulation (VR-Applikation) und Präsentation (Darstellung) benötigt der Applikationsentwickler keine Einzelheiten des unterliegenden Bildgenerierungssystems, sondern kann sich völlig auf die Entwicklung der VR-Applikation konzentrieren. Das System stellt Werkzeuge bereit, um den Script-Code zur Laufzeit zu verändern, und somit das Verhalten der Applikation interaktiv zu entwickeln und anzupassen. Die Ergebnisse dieser Änderungen sind mit minimaler Verzögerung erkennbar, ohne das das System neu starten oder laden muss.

Das Alice-System (siehe Abbildung 2.1) besteht aus zwei voneinander getrennten Prozessen, die auf verschiedenen Prozessoren oder Rechnern laufen: dem Simulationsprozeß und dem Renderingprozeß. Die Renderingframerate ist somit unabhängig von der Simulationsframerate. Der Vorteil ist, dass rechenintensive Simulationen, die eine niedrige Framerate haben, nicht die Renderingframerate beeinflussen. Somit kann der Renderingprozeß auch bei einer sehr komplexen Applikation in Echtzeit arbeiten und der Benutzer, der z.B. Position oder Orientierung verändert, bekommt in Echtzeit Bilder aus dem neuen Blickwinkel.

Der Renderingprozeß ist verantwortlich für das Ansteuern der externen Ein-

und Ausgabegeräte. Sobald die Simulation startet, wird automatisch der Renderingprozeß gestartet. Bei stereoskopischer Ausgabe starten zwei Renderingprozesse optional auf unterschiedlichen Rechnern. Zusätzlich wird ein weiterer Prozess erzeugt, der die Daten vom Eingabegerät an den Simulations- und Renderingprozeß weiterreicht. Alle Veränderungen der lokalen Szenerie werden am Ende des aktuellen Frames an den Renderer in Form von Nachrichten (commands) weitergegeben, so dass dieser seine lokalen Daten abgleichen kann. Unterstützte Eingabegeräte sind u.a. Trackinggeräte mit 6 Freiheitsgraden wie der Polhemus Fasttrack, Datenhandschuhe (Powerglove), sowie traditionelle Eingabegeräte wie Tastatur und Maus.

Das Renderingsystem unterstützt sowohl OpenGL als auch DirectX. Es können verschiedene virtuelle Kameras definiert werden, die jeweils in unterschiedlichen Ausgabefenster dargestellt werden.

Der Simulationsprozeß interpretiert das vom Applikationsautor geschriebene Python-Skript und aktualisiert dementsprechend die lokale Applikationsdatenbank. In jedem Frame werden die Methoden der benutzten Alice-Objekte aufgerufen und abgearbeitet. Mit diesen Methodenaufrufen wird der interne Status des einzelnen Objekts wie Position, Orientierung und Geschwindigkeit berechnet und verändert. Am Ende jedes Simulationsschrittes werden die Veränderungen der Szene als Nachrichten an den Renderingprozess übermittelt.

Eine dreidimensionale Szene wird als hierarchischer Graph (Szenengraph, siehe Abbildung 2.2) modelliert, in welcher jedes Objekt mehrere Unterobjekte als Kinder verwaltet und zusammenfasst. Die Transformation einzelner Objekte ist wie in allen Szenengraphen standardmäßig relativ zu dem Koordinatensystem seines Vater-Objekts. In Alice können Transformationen aber auch beliebige andere Referenzierungsobjekte benutzen. So ist es möglich, ein neues Objekt einfach in die Szene zu platzieren, indem man es beispielsweise einen Meter vor der virtuellen Kameraposition platziert, unter Einbeziehung der Blickrichtung des Benutzers. Alice führt damit einen neuartigen interessanten Beziehungstyp ein.

VR-Applikationen in Alice werden mit zwei Werkzeugen erzeugt: für Geometriedaten wird das CAD-Programm SilverScreen [28] benutzt. Die exportierten Daten werden von Alice gelesen und daraus hierarchischen Geometrien (Szenengraph) erzeugt. Die Spezifizierung der dynamischen Elemente erfolgt anschließend in Python für jeden einzelnen Aspekt von Hand. Zusätzlich ermöglicht das System erfahrenen Anwendern, eigene Erweiterungen in C oder C++ zu schreiben, die statisch oder dynamisch zu dem System hinzugelinkt werden.

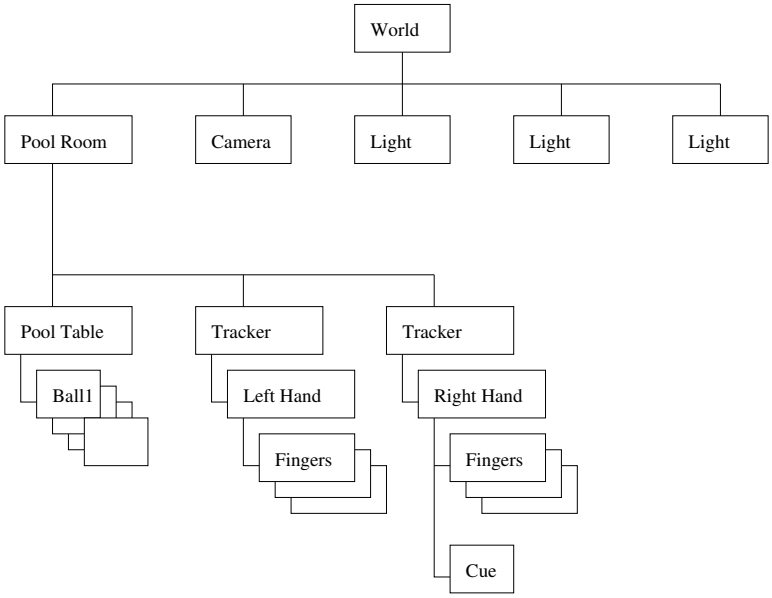

Abbildung 2.2: Szene organisiert als Graph in Alice

Eine im System bereits vorhandene Anzahl an Objekten erlaubt es, geometrische Objekte zu erzeugen und zu manipulieren. Wie bei Systemen, die auf Szenengraphen aufbauen allgemein üblich, werden neu erzeugte Objekte im Szenengraph an einen Elternknoten angehängt und übernehmen dessen lokales Koordinatensystem. Dadurch ist es möglich, mit wenigen Befehlen viele Objekte gleichzeitig zu beeinflussen. Besteht eine Szene z.B. aus einem Billardtisch mit mehreren Billardkugeln auf diesem Tisch, führt ein einfaches Verschieben des Tisches dazu, dass die Kugeln mit dem Tisch mitbewegt werden. Der Szenengraph von Alice kann jederzeit reorganisiert werden; soll zum Beispiel eine Billardkugel vom Tisch auf den Boden fallen, genügt ein Befehl und es wird das die Kugel repräsentierende Objekt lokal an den Knoten umgehängt, der den Boden repräsentiert.

2.2.2 Avacado / Avango

Avango [135][134][123] ist ein von der GMD in C++ entwickeltes VR-Framework für verteilte VR-Anwendungen.

Avango benutzt ebenfalls einen Szenengraph, um die virtuelle Umgebung zu repräsentieren. Der Szenengraph basiert auf Strukturen, die von der IRIS Performer [111] Bibliothek zur Verfügung gestellt werden. Performer selbst ist ein reines System zur Darstellung von 3D-Szenen wie zum Beispiel OpenSG [65]. Performer

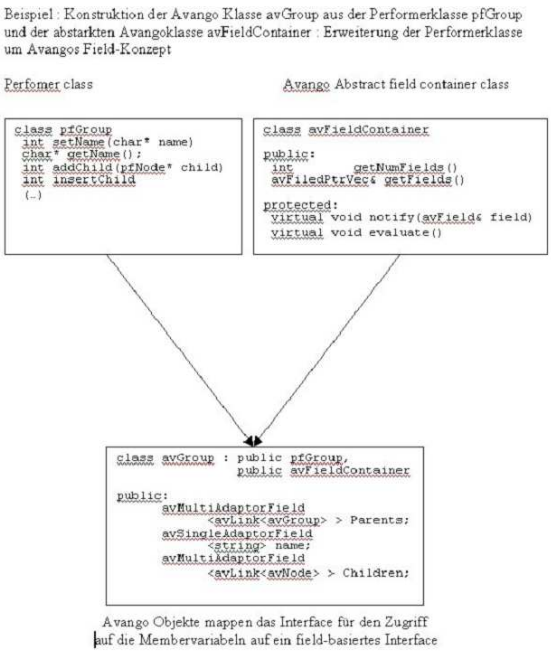

Abbildung 2.3: Avango-FieldContainer als Ableitung der Perforce-Klasse

beinhaltet keine Funktionalitäten, um weitere Ein- und Ausgabegeräte anzusprechen, Animationen abzuarbeiten oder zum Beispiel Sound auszugeben. Die Knoten des Graphen werden dem Anwendungsentwickler als relative simple C++ Objekte zur Verfügung gestellt. Das Erzeugen und Löschen von Knoten und die Manipulation der Knotenparameter wird über C/C++ Standardfunktionen ermöglicht.

Avango-Objekte sind grob in zwei Kategorien einteilbar: die Knoten der Szene und die Sensoren. Während Knoten dazu dienen, komplexe Objekte im Szenengraphen zu repräsentieren und zu rendern, sind die Sensoren die Schnittstelle zur 'realen Welt', mittels derer diverse Eingabegeräte abgefragt werden. Alle Objekte sind sogenannte 'Fieldcontainer', d.h. sie speichern alle objektspezifischen Zustandsinformationen in Feldern. Um dies zu ermöglichen, werden per Mehrfachvererbung Performer-Objekte mit Avangos Feldcontainer-Konzept vereinigt. Dadurch haben alle Objekte ein einheitliches Interface für den Zugriff auf diese Felder (siehe Abbildung 2.3).

Diese Felder können, soweit sie kompatibel zueinander sind, miteinander verknüpft werden. Somit entsteht ein Datenflussgraph, der zur Beschreibung von interaktiven und dynamischen Elementen eingesetzt wird. Die verknüpften Felder

leiten Änderungen ihrer Werte sofort weiter, wobei Zyklen erkannt und aufgelöst werden. Sobald ein Feld einen neuen Wert erhält, wird die notify-Funktion des zugehörigen Fieldcontainers aufgerufen, die auf diese Wertänderung hinweist. Der Fieldcontainer kann auf diese Änderung reagieren und gegeben falls weitere Felder manipulieren. Nachdem alle Nachrichten an alle Fieldcontainer für den aktuellen Frame weitergeleitet sind, wird die evaluate-Funktion der Fieldcontainer aufgerufen, die mindestens eine Wertänderung in einem zugehörigen Feld hatten. Diese Funktion erlaubt es den Fieldcontainern Aktionen durchzuführen, die von mehr als nur einem geänderten Feld abhängig sind.

Sensoren sind die Schnittstelle von Avango zur 'realen Welt'. Sie kapseln den Code der benötigt wird, um auf beliebige Eingabegeräte zuzugreifen. Da sie nicht direkt vom Performer abgeleitet werden, können sie nicht in den Szenengraph integriert werden. Sobald ein Eingabegerät neue Daten bereitstellt, werden die entsprechenden Felder des zugehörigen Sensors mit diesen gefüllt und anschließend über die Feldverknüpfungen von dem Sensor an die Knoten im Szenengraphen weitergeleitet.

Avango unterstützt verteilte Anwendungen indem der Szenengraph transparent für den Entwickler auf allen Rechnern synchronisiert wird. Da aus Performance-Gründen alle beteiligten Parteien den Szenengraphen vollständig lokal halten müssen, wird dieser als Kopie lokal auf den jeweiligen Rechnern dupliziert.

Durch die Skriptsprache Scheme steht dem Applikationsentwickler das komplette API des Systems zur Verfügung. Jedoch können nur globale Empfänger für Nachrichten des Systems oder einzelner Knoten angemeldet werden. Es gibt keine Möglichkeit den Scriptcode direkt als Teil einer Knotenbeschreibung in die Szene einzubetten.

2.2.3 dVS/dVISE

Die dVS Umgebung [47] ist ein kommerzielles System zur Entwicklung verteilter komplexer VR-Simulationen. Der Schwerpunkt liegt auf Produktsimulation und virtueller Produktentwicklung. dVs basiert auf einer Client-Server-Architektur und verschiedenen, auf mehreren Rechnern verteilten Diensten (Actors), die die Simulation und Visualisierung steuern.

dVS ist als Werkzeug gedacht, das verschiedende Dienste zur Verfügung stellt, ähnlich wie ein Betriebssystem, die es den (verteilten) Actors erlaubt, miteinander zu interagieren.

Das System bietet mehrere Standarddienste an, die die meisten Aufgaben

in einem VR-System abdecken. Zu diesen Diensten gehören ein Visual Actor (zuständig für das Rendern der Szene), ein Body Actor (Schnittstelle zwischen User Applikation und an das System angeschlossenen Geräten), ein Audio Actor, ein Tracker Actor (abstrahiert die Schnittstellen für verschiedene unterstützte Trackingsysteme) und ein Collision Actor.

Kernstück von dVS ist die Datenbank (VL), in der alle Objekte der aktuellen Anwendung verzeichnet sind und auf die von den einzelnen Actors zugegriffen werden kann, um Objekte zu erzeugen und zu manipulieren. Ein Objekt ist eine Datenstruktur mit Attributen (data fields) und Methoden (u.a. create, update, delete). Objekte werden manipuliert, indem die Actors verschiedene Events an die VL senden, die diese dann auswertet und an die für den jeweiligen Event registrierten Objekte weiterleitet.

Applikationen für dVS können auf zwei Arten entwickelt werden: mit dem VCToolkit, einer C-Programmierschnittstelle, die von der unterliegenden verteilten Architektur abstrahiert und es dem Entwickler erleichtern soll, sich auf die Anwendung und nicht auf das System zu konzentrieren, und der high-level Skriptsprache dVISE, die auch Nichtprogrammierern einen leichten Umgang mit dVS ermöglichen soll.

Anwendungen mit dem VCToolkit werden erstellt, indem der Programmierer Objekte erzeugt und diesen Events zuweist. Über Callback-Funktionen, die beim System registriert werden, kann dann auf diese Events reagiert werden. Das Eventhandling übernimmt dabei das Toolkit. Zusätzlich besteht noch die Möglichkeit, Timer callbacks zu definieren, die zu bestimmten Zeitpunkten ausgelöst werden und als Events an die sich dafür angemeldeten Objekte weitergeleitet werden.

Im Gegensatz zum VCToolkit, das primär für C/C++-Entwickler gedacht ist, bietet dVS mit dVISE eine Schnittstelle für Benutzer, die VR-Applikationen entwickeln wollen, ohne dabei selbst programmieren zu müssen. dVISE ist ein Application Actor, der auf dem VCToolkit aufsetzt und über simple Script-Textfiles gesteuert wird.

2.2.4 Lightning

Das VR-System Lightning [19][20] ist ein vom Fraunhofer Institut für Arbeitswissenschaft und Organisation entwickelter Prototyp für ein flexibles Virtual Reality System.

Auf unterer Ebene existieren verschiedene Module, um die Ein- und Ausgabe unter einem Abstract Device Layer zu implementieren. Dazu gehören der Visual

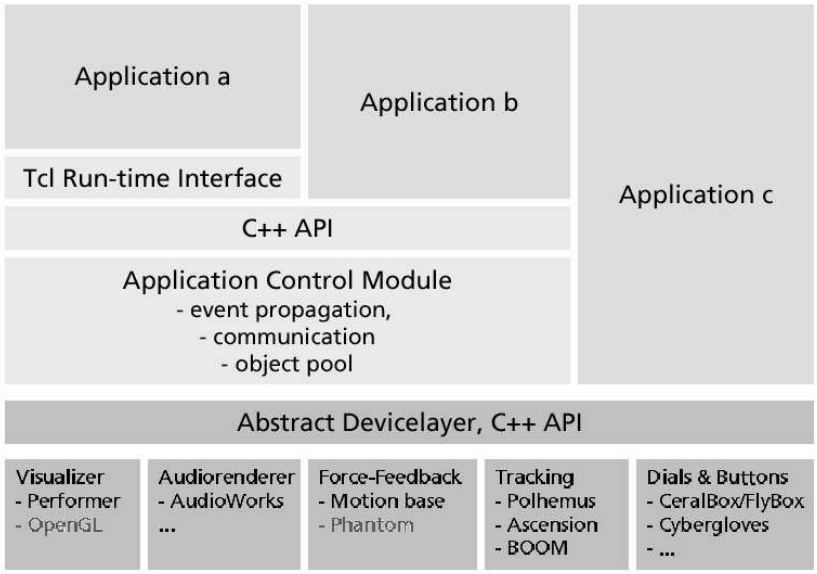

Abbildung 2.4: Lightning Systemarchitektur

Renderer, der Audio Renderer, das Force-Feedback-Modul, die Unterstützung für Trackinggeräte und andere Eingabegeräte.

Kernstück von Lightning ist der Objektpool, der im Application Control Module über dem Abstract Device Layer angesiedelt ist. Applikationen werden erstellt indem Objekte erzeugt, manipuliert und gelöscht werden. Interaktion und Verhalten der Applikation werden zum einen über Skriptsprachen (TCL) und zum anderen über ein Event-Propagation-Konzept implementiert, das sich an VRML bzw. X3D [143] anlehnt. Dieses Event-Propagation-Modell sorgt für die Kommunikation der einzelnen Objekte untereinander, indem es sie über sogenannte routes miteinander verbindet.

Objekte in Lightning haben ein standardisiertes Interface für die Kommunikation untereinander. Sie besitzen spezielle Felder, sogenannte slots, die zur Eventbehandlung und -weiterleitung benutzt werden. Die Objekte und ihre Verbindungen untereinander ergeben einen gerichteten Graphen, der einmal pro Frame der Applikation ausgewertet wird. Angefangen bei Sensoren, die Daten in den Graphen hineinleiten (Eingabesensoren, Zeitsensoren, etc.), werden die Verknüpfungen der Objekte verfolgt und die Daten entsprechend an ihr Ziel weitergeleitet. Diese Zielobjekte können daraus unter Umständen neue Daten für andere Objekte produzieren, die weitergegeben werden. Das System besitzt eine einheitliche Update-

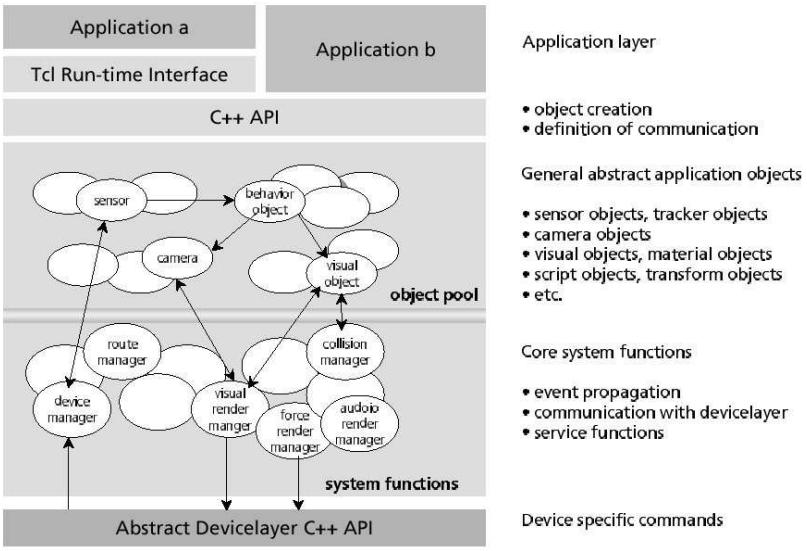

Abbildung 2.5: Objektpool zur Organisation der Szeneelemente in Lightning

Funktion, die dazu dient, das Objekt durch Auswertung der in den slots eingetroffen Daten auf den neuesten Stand zu bringen, und seine interne Repräsentation (state) dementsprechend anzupassen. Diese Update-Funktion wird nur ausgeführt, falls tatsächlich neue Daten beim Objekt eingetroffen sind. Wenn alle Verbindungen und der daraus resultierende Informationsfluss abgearbeitet ist und keine Daten mehr zu verteilen sind, werden die Informationen für die verschiedenen Rendering-Subsysteme (Video, Audio) aufbereitet. Erst dann kann der nächste Schritt der Simulation erfolgen .

Um die Objekte und Events im System zu steuern, existieren verschiedene Kontrollmodule (managers). Der 'object manager' kümmert sich um das Erzeugen und Löschen der Objekte in der Applikation. Der 'route manager' ist für die korrekte Weiterleitung der Events und der daraus resultierenden Daten zuständig, während der 'device manager' die existierenden Eingabegeräte überwacht und abfragt.

2.2.5 Maverik

Maverik [58][57][59] steht für Manchester Virtual Environment Interface Kernel, ein VR-System für large-scale Industrie-Applikationen und virtuelle Umgebungen,

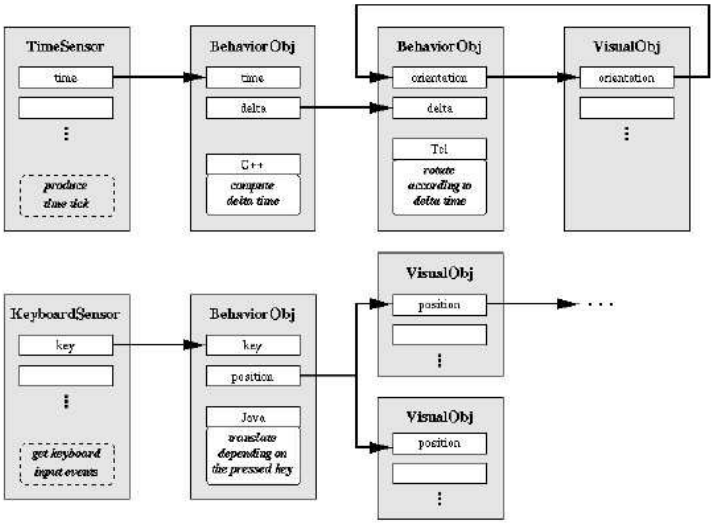

Abbildung 2.6: Das Propagieren von Nachrichten in Lightning

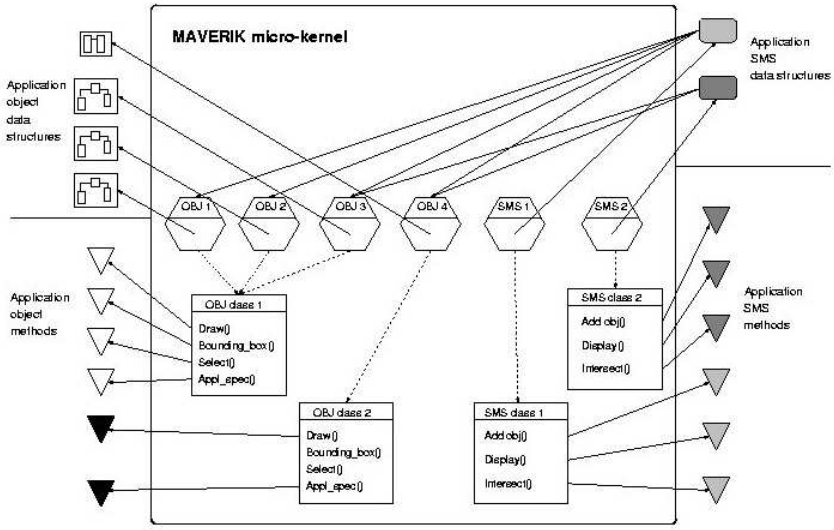

Abbildung 2.7: Maverik Systemarchitektur

das am Department of Computer Science der University of Manchester entwickelt wurde.

Maverik ist ein in C geschriebener Micro-Kernel, der in Anlehnung an das X-Window-System die grundsätzlichen Funktionen eines VR-Systems bereitstellt. Es ist wie ein Windowsystem aufgebaut, das anstatt 2D-Fenster 3D-Welten verwaltet: Marverik besteht aus einer Anzahl an Bibliotheken, die die wichtigsten Funktionen bereitstellen. Wie die gängigen 2D-Frameworks haben Marverikprogramme eine Eventschleife, in die sich Applikationen durch die Definition und die Bereitstellung von Callbacks einklinken kann. Sämtliche applikationsspezifischen Daten sind aus dem Kernel ausgelagert (Geometrien, Organisation der Daten, Verhaltensbeschreibungen und deren Implementierung).

Der Kernel definiert Klassen von Daten und Objekten, die aus Variablen und Methoden bestehen. Für alle Methoden existieren default- Implementationen. Es ist dem Applikationsentwickler möglich die Default-Methoden zu benutzen, oder bei Bedarf diese durch eigene Implementationen zu ersetzen. Die neuen Objekte und Methoden werden beim Kernel registriert.

Der Kernel ist minimalistisch designt, so dass es aufwendig ist eine neue Applikation zu implementieren, die nur auf die vom Kernel bereitgestellten Features zurückgreift. Um den Aufwand zu verringern, existieren mehrere Hilfsfunktionen, die in verschiedenen Bibliotheken untergebracht sind. Diese können je nach den Anforderungen der Applikation von den Anwendungen benutzt, angepasst oder ersetzt werden. Die Hilfsfunktionen sind verschiedenen Aufgabenbereichen zugeteilt: Spatial Management System (SMS), Culling und Level of Detail Processing, Constraintbased Manipulation, Eingabe, Navigation, Kollisionserkennung, Applikationscode und externe Fileformate.

Applikationen in Maverick werden in C entwickelt. Die Applikation definiert ihre Daten und stellt Funktionen über Callbacks bereit, über die der Kernel auf die Daten zugreifen und sie manipulieren und darstellen kann. Für die gängigen in VR-Systemen vorkommenden Aufgaben bietet der Kernel bzw. die mitgelieferten Hilfsbibliotheken default-Implementationen, die von der Anwendung genutzt werden können. Bei Bedarf können diese Defaultmethoden durch eigene, besser geeignete oder effizientere Methoden ersetzt werden. Das ist im Vergleich zu anderen Systemen mit relativ viel Aufwand verbunden, bietet aber ein hohes Maß an Flexibilität. Der Entwickler hat die Möglichkeit, die für seine Applikation effizienteste Implementierung zu benutzen, da das System keinerlei Einschränkung hinsichtlich der internen Repräsentation der benutzten Daten und der mit diesen

Abbildung 2.8: Beispielanwendung in Maverik

Daten arbeitenden Funktionen macht. Dies ist allein Aufgabe des Anwendungsentwicklers.

Um auf Eingabegeräte zugreifen zu können, existiert im Kernel ein Mechanismus, mit dem externe Geräte wie Maus, Keyboard, Trackingsysteme mit 6 Freiheitsgraden oder auch Spracherkennung unterstützt werden können. Dazu stehen bis zu drei Callbackmethoden zur Verfügung, die entweder von der Applikation selbst oder, soweit bereits vorhanden, von dem Navigationsmodul bereit gestellt werden. Die erste der drei Callbackmethoden ermittelt die Position des Eingabegerätes in dessen lokalem Koordinatensystem, z.B. Pixelkoordinaten relativ zur Position des aktuellen Fensters im Fall der Maus. Die zweite Callbackmethode überträgt die zuvor ermittelten lokalen Koordinaten in das globale Koordinatensystem der Anwendung. Die dritte Callbackmethode dient zur Überwachung von Ereignissen die das Eingabegerät auslöst, wie z.B. das Drücken von bestimmten Tasten an dem Gerät. Welche dieser drei Callbacks implementiert werden, hängt von der Art des Gerätes ab: bei einer Maus oder einem Tracker sind alle drei Methoden notwendig, Keyboard oder Spracherkennung benötigen lediglich die dritte Methode, da die ersten Beiden für diese Art Gerät nicht definiert sind.

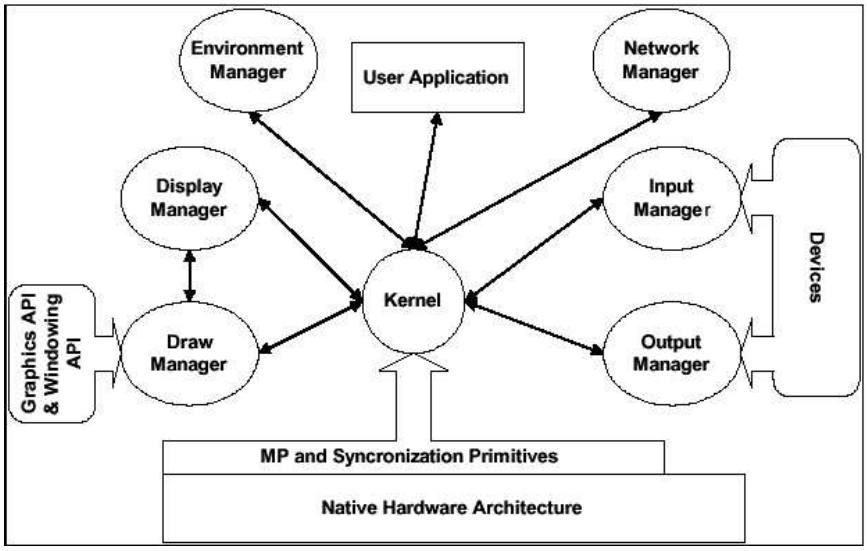

Abbildung 2.9: VR-Juggler Systemarchitektur

2.2.6 VR-Juggler

VR-Juggler [17][18][103] ist eine VR-Library, die an der Iowa State University mit dem Ziel entwickelt wurde, in portables und skalierbares VR-Entwicklungssystem zur Verfügung zu stellen. Der VR-Juggler bietet Unterstützung für CAVE-artige Projektionssysteme, HMDs, Trackingsysteme, 3D-Maus und Cybergloves. Das System ist unterteilt in verschiedene Module (managers), die sich um genau definierte Aufgaben wie Input, Network, und Display bzw. Ausgabe kümmern.

Der Kernel-Manager ist verantwortlich für die Kommunikation der einzelnen 'manager' untereinander und bietet ein Interface für die Applikationsentwicklung. Durch diese Struktur können einzelne Module (wie zum Beispiel das Renderingsystem) mit minimalen Auswirkungen auf das restliche System ausgetauscht werden. Die Applikationen haben ausschließlich über den Kernel-Manager Zugriff auf das System.

Ein- und Ausgabemodule werden hinter generischen aber klassifizierten Interfaces versteckt, so dass neue Geräte durch Implementierung dieser Schnittstellen in das System integriert werden können. Weitergehende Unterstützung für Benutzerinteraktionen existieren jedoch nicht. Entwickler müssen die Eingabegeräte selbst überwachen (kein Event-Konzept), indem sie entsprechenden C++-Code für ihre VR-Applikationen schreiben. Es existiert keine Systemunterstützung für Kollisionserkennung oder Objektverhaltensbeschreibungen. Diese müssen bei Bedarf

Abbildung 2.10: Statische Unreal Szenendarstellung

von der Applikation bereit gestellt werden. Applikationen werden in C++ geschrieben. Das System sieht keine Skriptsprache oder andere, intuitivere Möglichkeiten der Applikationsentwicklung vor.

2.2.7 Unreal Engine

Unreal Engine [130][106][129] ist ein kommerzielles Produkt von epic Games, Inc. zur Entwicklung von 3D-Computerspielen aus der Perspektive des Spielers mit Schwerpunkt auf Mehrspielermodi über Local-Area-Networks und Internetverbindungen. Das System stellt für jede Spieleumgebung einen Server bereit, mit dem die einzelnen Clients verbunden werden. Im Einzelspielermodus laufen Client und Server auf demselben Rechner, während im Mehrspielermodus das Spiel auf einem dedizierten Server läuft, an dem sich die Clients einloggen.

Jedes Spiel läuft in einem 'Level' ab. Ein Level besteht aus Geometriedaten und Actors. Ein Actor wird entweder vom Spieler oder von einem Script gesteuert; im letzteren Fall definiert das Script, wie sich dieser Actor bewegt und mit anderen Objekten interagiert. Der Server ist dabei für die Applikationslogik (das Spiel) verantwortlich. Er berechnet den Ablauf des Spiels und sendet die Daten an die

Abbildung 2.11: Unreal Szene mit Interaktiven Element

Clients weiter. Die Clients rendern die Szene, fragen die Eingabegeräte (Keyboard, Maus, Joystick) ab und schicken diese Daten zurück an den Server.

Zentral sind die Begriffe des 'Actors' und des Zustands ('game state'). Ein Actor ist ein Objekt, das auf unterschiedlichen Arten den Zustand eines beliebigen anderen (oder eigenen) Objektes verändern kann wie zum Beispiel die vom Spieler gesteuerten Figuren, vom Computer berechnete Gegner des Spielers oder auch Objekte, die irgendwie beeinflusst werden können. Unter Zustand ('game state') eines Levels wird die Liste aller im Spiel befindlichen Actors und die Werte der diesen zugehörigen Variablen zu einem bestimmten Zeitpunkt verstanden.

Um den Ablauf eines Spiels zu kontrollieren, teilt die Engine jede Sekunde in mehrere gleich lang dauernde Frames ein, die normalerweise zwischen 1/100 und 1/10 Sekunde lang sind. Dies ist die kleinste Zeiteinheit (deltaTime) innerhalb der alle Actors upgedated werden. Die deltaTime hängt direkt von der Geschwindigkeit der CPU ab – je schneller diese ist, desto kürzer ein Frame.

Die Update-Schleife von Unreal Engine besteht aus drei zentralen Teilen:

- Der Server sendet den aktuellen Zustand des Spiels an die Clients.

- Die Clients übertragen ihre Änderungswünsche an den Server, empfangen die neuen Spielinformationen von diesem und rendern die aktuelle Szene.
- Sobald deltaTime Zeit verstrichen ist ruft jeder der Server die tick-Funktion auf, die den neuen Status/Zustand des Spiels berechnet. Eine Zustandsänderung ist dabei das Erzeugen oder das Löschen eines Actors oder das Verändern einer Variable eines Actors.

Die tick-Funktion bringt alle Actors in der Szene auf den aktuellen Stand, informiert sie über sie betreffende Events im Spiel und führt bei Bedarf Scriptcode aus.

Hauptsächlich wird die Unreal Engine für Spiele und Anwendungen auf verteilten Systemen über das Internet genutzt, wo z.T. noch 28.8K Modems vorhanden sind, die vom Spiel unterstützt werden. Da die Menge an Daten, die zu einer vollständigen Übermittlung aller Zustandsinformationen des Spiels an jeden Client nötig wären die Bandbreite der meisten Spielteilnehmern bei weitem übersteigt, werden nur die für die Darstellung beim Client relevanten Informationen an ihn übertragen. Daten von Actors, die außerhalb des Einflussbereiches der aktuellen Szene liegen, sind überflüssig und werden nicht versendet.

Zur Applikationsentwicklung stehen zwei Methoden zur Verfügung: Geometriedaten wie das Layout der Levels oder neue Modelle für Spielfiguren werden mit Tools wie 3D-StudioMax und dem mitgelieferten UnrealEditor generiert. Die eigentliche Applikationslogik wird mit UnrealScript erstellt, kompiliert und in das System eingebunden.

UnrealScript ist eine objektorientierte Sprache, die stark an Java orientiert ist, aber einige für Spieleprogrammierung nützliche Unterschiede bzw. Erweiterungen besitzt. In erster Linie sind das die einfache Unterstützung von zeit- und eventbasierten Zustandsdefinitionen und -veränderungen. Großer Wert wurde auf die einfache Erweiterbarkeit des Spiels gelegt: durch die Vererbungstechnik können vorhandene Klassen abgeleitet und verändert werden. Die Möglichkeiten reichen vom einfachen Ändern eines bestimmten Parameters im Spiel bis zur Definition von komplett neuen Welten.

Unreal nutzt verschiedene Möglichkeiten Dynamik in die Level zu bringen: benutzer- und eventbasiert. Die eventbasierte Dynamik wird mittels Triggern realisiert, die auf bestimmte Zustandsveränderungen reagieren. Dazu zählen Trigger, die auf die Nähe eines (bestimmten oder unbestimmten) Objektes reagieren, sobald die Entfernung zwischen diesen einen definierten Schwellenwert unterschritten hat oder weitere, die auf die Kollision mit einem anderen Objekt reagieren. Eine be-

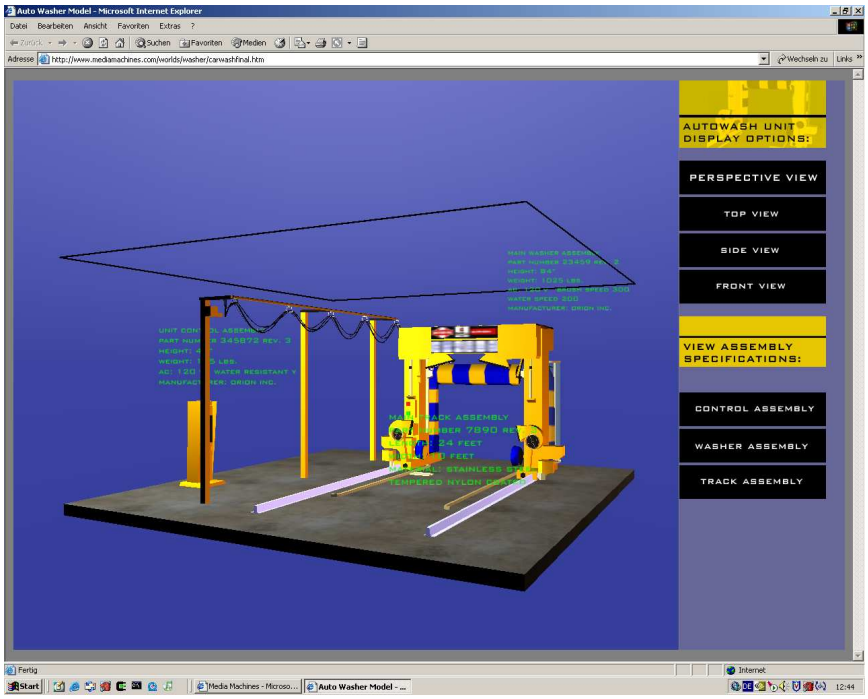

Abbildung 2.12: X3D-Anwendung im Flux-Player

sondere Art dieser Trigger ist der TimeTrigger, der zu einer bestimmten Zeit nach Beginn des Spiels ausgelöst wird; entweder einmalig oder in periodisch gleichen Abständen.

2.2.8 Flux und X3D-Systeme

Flux [87] ist ein Visualisierungssystem für X3D-Szenen [143]. Das System soll an dieser Stelle als Beispiel für ein typisches modernes X3D-basiertes System stehen. Das System hält sich streng an den ISO-Standard und setzt diesen nahezu komplett um. Somit kann es alle im Standard definierten Knoten verarbeiten und bietet mit dem SAI-Interface eine standardkonforme Schnittestelle für Scripte und externe Anwendungen. Das System ist als Web-Browser-Plugin konzipiert und erlaubt dem Anwender lokale und auf Web-Servern vorliegende 3D-Szenen zu verarbeiten (siehe Abbildung 2.12). Es ermöglicht dem Benutzter komplexe Welten akustisch und visuell zu erleben und interaktiv zu steuern. Es beinhaltet jedoch keine Werkzeuge, diese zu erstellen. Dazu muss der Entwickler Werkzeuge einsetzen die X3D-Szenen erzeugen können, wie zum Beispiel Vizx3D [138].

X3D ist ein ISO-Standard zur Beschreibung von interaktiven 3D-Szenen, der gleichermaßen für Internetanwendungen, als auch für Intranets entworfen wurde. Ziel ist ein universelles Beschreibungsformat für virtuelle Welten und Szenarien, die aus multimedialen Inhalten, wie dreidimensionale Grafiken und Audio/Videobestandteilen, zusammengesetzt sind. Das Format beinhaltet die für heutige 3D-Anwendungen nötigen Grundfunktionalitäten wie hierarchische Transformationen, Lichtquellen, Kamerapositionen, Geometriebeschreibungen, Animationen, Nebel, Materialeigenschaften der Komponenten und Texture mapping. X3D kommt in unterschiedlichen Umgebungen zum Einsatz wie zum Beispiel industrielle und wissenschaftliche Forschung und Entwicklung, Multimediapräsentationen, Unterhaltung, Internetseiten und verteilte virtuelle Welten.

X3D ist in der Lage, statische und dynamische 3D- und Multimedia-Objekte mit Verbindungen zu anderen Medien wie Text, Audio, digitale Filme und Bilder darzustellen. Grundsätzlich definiert ein X3D-Dokument einen Baum von Objekten, den sogenannten Knoten. Diese Knoten sind hierarchisch strukturiert und beinhalten Ein- und Ausgabeschnittstellen über sogenannte *Slots.* Jeder Slot kann entweder Nachrichten empfangen und versenden. Zur Modellierung von dynamischen Veränderungen verbindet der Anwendungsentwickler jeweils einen Ausgabe-Slot mit einem Eingabe-Slot vom gleichen Datentypen mit einer *Route.* Neben Knoten mit Standarttypen kann der Entwickler Script-Knoten mit anwendungsspezifischen Schnittstellen anlegen. Das Verhalten der Scripts (Reaktion auf Nachrichten, Erzeugung von Ausgaben) sind in ECMAScript oder Java programmtechnisch zu beschreiben.

X3D ist eine Erweiterung des VRML ISO Standards und beinhaltet alle Strukturen und Knotentypen der ursprünglichen Spezifikation. Es erweitert den VRML Standard vor allem in drei Bereichen:

Knotentypen Neue Knotentypen die es dem Entwickler erlauben neuartige Features von Graphik-Hardware zu nutzen (zum Beispiel Multitexture), sowie neue Knotentypen, um weitere Anwendungsfelder zu erschließen (zum Beispiel NURBS Flächenbeschreibungen).

Kodierung VRML spezifizierte nur ein einzige von Inventor [127] abgeleitete text-basierte Kodierung der Welten. Die X3D-Spezifikation beinhaltet drei unterschiedliche Kodierungen: VRML-Syntax, XML-basiert und Binärdaten.

Schnittstellen X3D vereinheitlicht mit der SAI (Scene Access Interface) den Zu-

griff und Manipulation der Szenendaten auf Script Ebene und durch externe Programme.

Im Großen und Ganzen ist X3D eine modernisierte Version von VRML. Viele Aspekte der VRML Spezifikation waren nicht eindeutig und sind in der X3D Spezifikation ausdrücklich aufgenommen. Weiterhin wurde versucht einige globale Ziele umzusetzen, um die Akzeptanz gegenüber VRML weiter zu erhöhen:

Erweiterbarkeit Das Format soll nicht im Standard vorgesehene Erweiterungen und Elemente einfach erzeugt und integrieren können.

Einfachheit der Entwicklung von Applikationen Die Sprache soll es externen Programmen/Werkzeugen einfach machen, X3D-Dateien zu erstellen und zu bearbeiten. Ein weiteres Ziel ist die möglichst einfache Konvertierung von existierenden 3D-Formaten von und nach X3D.

Komposition Kombination Bestehende Objekte in einer X3D-Anwendung sollen in der selben Anwendung oder in einer Anderen einfach wiederverwendet werden können. Das X3D-Format soll so gestaltet sein, dass neue Applikationen aus bereits bestehenden Objekten zusammengesetzt werden können.

Plattformunabhängigkeit Ein existentiell wichtiges Kriterium ist, die Einsatzmöglichkeit auf einer großen Bandbreite verschiedener verfügbarer Plattformen zu sichern.

X3D ist in der Lage, statische und dynamische 3D- und Multimedia-Objekte in Verbindung zu anderen Medien wie Text, Audio, digitale Filme und Bilder darzustellen. Werkzeuge zum Erstellen und Anzeigen von X3D-Applikationen sind in einer großen Anzahl auf den unterschiedlichsten Plattformen verfügbar. Grundsätzlich definiert ein X3D-Dokument einen Baum von Objekten, sogenannten Knoten. Diese Knoten sind hierarchisch strukturiert und können sich gegenseitig beeinflussen. Die Hierarchie von Knoten wird im Allgemeinen Szenengraph genannt.

2.3 Zusammenfassung

An dieser Stelle soll eine kurze tabellarische Zusammenfassung (siehe Tabelle 2.1) in Form einer Auflistung der positiven und negativen Eigenschaften und

	Alice	Avango	dVise	Ligthning	Maverik	VRJ	Unreal	Flux
Ereigniskanten	-	+	-	+	-	- -	-	++
Scripting	++	+	+	++	-	- -	+	+
Skalierbarkeit	-	+	-	+	-	-	-	-
Geräteunterstützung	-	+	+	+	+	+	-	- -
Manipulatoren	+	-	-	-	-	-	-	+
Animation	+	+	-	+	-	- -	+	+

Tabelle 2.1: Klassifikation der Eigenschaften der untersuchten VR-Systeme

Fähigkeiten der zuvor beschriebenen Systeme erfolgen. Daraufhin werden Anforderungen für ein neuartiges System abgeleitet.

Ereigniskanten Wieweit unterstützt das System den Nachrichtenaustausch zwischen Knoten über Ereigniskanten. Diese Eigenschaft ist die Grundlage für moderne *Flow-based-programming [95]* Architekturen.

Scripting Die Eigenschaft, Teile der Applikation in eine Script-Sprache umzusetzen. Diese erlaubt dem Anwendungsentwickler zeitunkritisches Verhalten plattformunabhängig und effizient zu programmieren.

Skalierbarkeit Wieweit lässt sich das System auf unterschiedlichsten Plattformen einsetzen und wieweit nutzt das System die gegebenen Resourcen optimal aus.

Geräteunterstützung Ist es möglich, beliebige mehrdimensionale Ein- und Ausgabegeräte anzuschließen, die vor allem in immersiven Umgebungen zum Einsatz kommen.

Manipulatoren Unterstützt das System High-Level Sensoren, mit denen der Entwickler interaktive Regionen in den Szenen unabhängig von den Eingabegeräten definieren kann.

Animationen Wieweit unterstützt das System Standardanimationen, wie zum Beispiel *Key-Frame* Interpolationen für Transformationen und Koordinaten.

Die Anforderungen an Rahmensysteme für VR (Virtual Reality) und AR (Augmented Reality) haben sich über die letzten 15 Jahre stark verändert. Das Design der hier aufgeführten Systeme, war auf einzelne Applikationsgruppen und Laufzeitumgebungen zugeschnitten und vorwiegend durch Fragestellungen der Computergrafik geprägt. Durch den immer breiteren Einsatz von VR- und AR-Technologien

in den unterschiedlichsten Anwendungen und der rasanten Entwicklung der Grafiksysteme in den letzten Jahren muss sich ein modernes Rahmensystem neuen Aufgaben stellen. Diese Bereiche werden hier kurz aufgelistet und in den nachfolgenden Kapiteln genauer untersucht und Lösungen entwickelt.

Effiziente Modellierung von Applikationslogik und -dynamik

Neben klassischen, vorwiegend statischen VR-Anwendungen (z.B. Walkthrough) muss das Rahmensystem dynamische Beschreibungen der VR-Welten in unterschiedlichen Formen unterstützen, synchronisieren und verarbeiten. Neben einfachen Logikmodellen zur Ablaufsteuerung muss die Umgebung Abstraktionen für komplexe Unterräume (z.B. Simulatoren und Automaten) bereitstellen. Dazu sollte ein effizientes und einheitliches Modell zur Modellierung von Verhalten und Dynamik geschaffen werden.

Flexible Interaktion und Sensorik

In Hinblick auf die Handhabe von Ein- und Ausgabegeräten lassen sich die hier vorgestellten Systeme in zwei Gruppen einteilen. Systeme, die vorwiegend in immersiven Umgebungen zum Einsatz kommen, bieten Schnittstellen, die es dem Anwendungsentwickler erlauben, direkt auf die Daten der Geräte zuzugreifen. Sie bieten kaum Interaktionsmethoden an, die unabhängig von den Geräteklassen sind. Auf der anderen Seite, gibt es Systeme, die hochwertige Abstraktionen anbieten, aber auf der anderen Seite kaum Zugriff auf die Datenströme der Geräte ermöglichen.

Ein modernes VR-System sollte dem Entwickler ein flexibles Rahmensystem für Interaktionsaufgaben zur Verfügung stellen, das unterschiedliche Abstraktionen bereithält und anbietet.

Skalierbarkeit

Im Gegensatz zu den ersten VR-Anwendungen, die noch vorwiegend auf homogenen Graphiksystemen implementiert waren, benutzen moderne Anwendungen inhomogene Plattformen von unterschiedlichster Ausprägung (Workstation, PC oder Handheld). Das Rahmensystem sollte somit die Eigenschaften der Laufzeitumgebung für die Anwendung abstrahieren, aber dennoch die Systemkomponenten und Resourcen (z.B. multi-pipe, multi-processor) möglichst optimal ausnutzen, um ein vorgegebenes Ziel zu erreichen.

Kapitel 3

Effiziente Modellierung von dynamischen Anwendungen

Die Dynamik in MR-Applikationen beschreibt die anwendungsspezifische Veränderung der 3D-Welt. Diese Veränderungen werden entweder von externen (z.B. Benutzerinteraktionen) oder internen (z.B. zeitgebundenen) Ereignissen angestoßen. Die eigentliche VR-Applikationsentwicklung beschreibt den Prozess geeignete Mittel einzusetzen, um auf diese Ereignisse zu reagieren und Veränderungen zu steuern.

In diesem Kapitel wird ein neuartiges Modell von Graphen eingeführt, das dem Anwendungsentwickler ermöglicht, die Dynamik von MR-Applikationen effizient zu modellieren.

3.1 Grundlagen

Die Untersuchung im vorangegangenen Kapitel hat gezeigt, dass nahezu alle aktuellen VR- und AR-Systeme einen Szenengraphen [104] als Datenstruktur für alle visuellen Objekte und deren Attribute einsetzen.

Ein Szenengraph ist ein gerichteter, azyklischer Graph, der meist inhomogene Knoten verbindet. Diese unterschiedlichen Knoten lassen sich grob in zwei Gruppen einteilen: Knoten die als Blätter fungieren beschreiben im Allgemeinen sichtbare Objekte (z.B. ein Fahrzeug) oder deren Eigenschaften (z.B. Farbe und Textur). Innere Knoten gruppieren und transformieren ihre Kinderknoten und bilden somit eine Hierarchie von räumlichen Beziehungen. Jeder Knoten besitzt einen Typen und eine für den Knotentypen festgeschriebene Liste an Attributen und deren Aus-

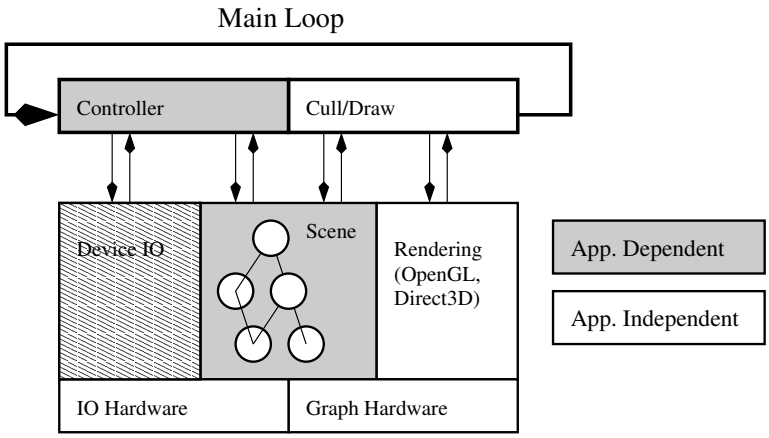

Abbildung 3.1: MR-Systemdesign mit applikationsspezifischer Kontrolleinheit

prägungen. Diese Typen sind im Allgemeinen nicht applikationsspezifisch, sondern werden von dem MR-System bereitgestellt. Die konkreten Knoten beschreiben somit eine 3D-Welt (die Szene), welche von dem Visualisierungssystem durch geeignete Traversierung als 2D-Bild synthetisiert wird.

Ziel der MR-Applikation ist es, diese Szenenbeschreibung immer wieder neu zu verändern, so dass der Benutzer den Eindruck einer interaktiven und dynamischen Welt erhält. Wenn man zur Vereinfachung die parallele Verarbeitung an dieser Stelle nicht berücksichtigt und sich auf einen Ausgabekanal beschränkt, ergibt sich daraus ein simples Applikationsmuster (siehe Abbildung 3.1).

Die Hauptschleife besteht aus der Steuereinheit (*Controller*), die die Welt verändert und dem Zeichner (*Drawer*), der die Welt abbildet. Der *Controller* übernimmt bei Bedarf neue Daten von den Eingabegeräten und möglichen Simulatoren und reagiert auf optionale zeitgesteuerte Veränderungen (z.B. Animationen). Er verändert dementsprechend den Szenengraphen, dessen Knoten und vor allem die Attribute der Knoten. Der Zeichner visualisiert daraufhin den aktuellen Zustand des Szenengraphen. Der Controller und der Aufbau des Graphen sind somit applikationsabhängig und müssen für jede Anwendung neu implementiert werden. Die Knotentypen sind dabei von dem Visualisierungssystem vorgegeben, jedoch sind die Ausprägungen der Instanzen und somit die Parametrisierung von Anwendung zu Anwendung verschieden.

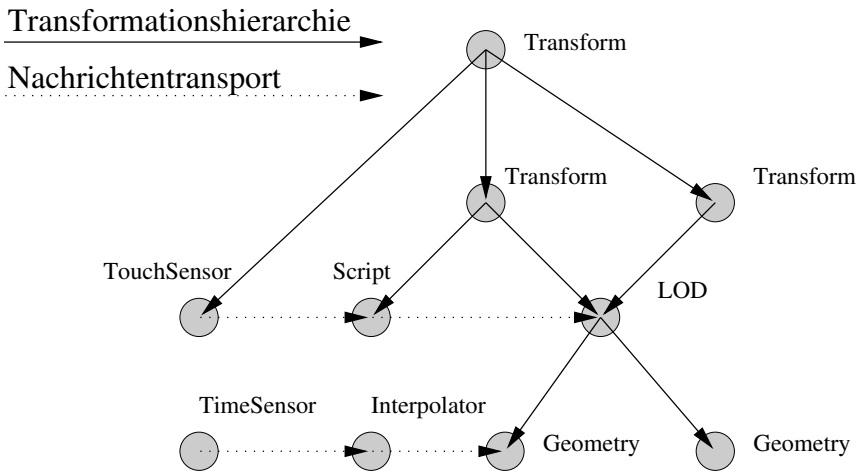

Abbildung 3.2: Kommunikationskanäle zwischen Knoten

3.2 Mehrdimensionale Graphen zur Modellierung der Dynamik

In X3D[143], Lightning[20] und Avocado[134] wurde der allgemeine, für VR-Systeme typische Ansatz des Szenengraphen erweitert. Die Systeme erlauben, orthogonal zu den hierarchischen Kanten weitere Nachrichtenkanäle zwischen den Knoten anzulegen. Diese Kanäle verbinden Aus- und Eingabe-Interfaces von unterschiedlichen Knoten. Diese Technik erlaubt es, Knoteninterfaces zu "verdrahten", und ist in der Komponententerminologie auch als "connection-oriented programming" [131] bekannt (siehe Abbildung 3.2). Diese Technik wird auch erfolgreich in anderen Bereichen, wie zum Beispiel in der Bildverarbeitung [70], zum Gestalten von 2D-Benutzerschnittstellen [21] oder als *Delegates&Events Pattern* in Programmiersprachen eingesetzt [82][77].

Die Anwendungsentwicklung besteht, ähnlich wie in allen VR-Systemen die einen Szenengraph benutzen, aus der Auswahl der geeigneten Knoten, deren Parametrisierung und der Festlegung der Beziehungen. Der Anwendungsentwickler bestimmt aber nicht nur die in den meisten Fällen statischen Hierarchiebeziehungen, sondern mit den Nachrichtenbeziehungen zwischen den Komponentenschnittstellen auch das dynamische Verhalten der Anwendung.

Dieses Design hat den entscheidenden Vorteil, dass es dem Applikationsentwickler erlaubt, neben den sichtbaren Objekten auch deren Verhalten zu modellieren, und somit die Dynamik der Anwendung zumindest teilweise zu bestimmen.

Jedoch haben die bisherigen Systeme zumindest drei entscheidende Nachteile:

Kein einheitliches Komponentenmodell Die Systeme setzen das eingeführte Komponentenmodell nicht systemweit ein, sondern nur für Elemente der Szene. Alle anderen Aspekte der Anwendung (z.B. Sensoren, Fenster, Prozesse) werden mit separaten Strukturen und Mechanismen abgebildet.

Beschränkung auf feste Kantentypen Die Systeme unterstützen Nachrichten und Hierarchiekanten und schaffen somit eine starre Struktur mit zwei Kantentypen. Sie generalisieren diesen Ansatz aber nicht um beliebige Kantentypen, die abhängig von ihren Typen evaluiert werden. Neuartige Beziehungstypen, die zum Beispiel Kanten einführen um physikalische Kräfte [136][24] oder *constrains* [56][35] abzubilden, sind nicht vorgesehen.

Beschränkung auf explizite Kanten Die Systeme unterstützen nur explizit angelegte Beziehungskanten. Fröhlich zeigt in seiner Arbeit [42], dass für die Modellierung von autonomen Objekten explizite Kanten ungeeignet sind. Deshalb definiert er in seiner Arbeit implizite Beziehungskanten, die zum Beispiel durch räumliche Abstände oder Typengleichheit generiert werden.

Aus diesem Grund wird hier ein neues Modell eingeführt: Jede Komponente besteht aus einer Parameterliste und einer Verhaltensbeschreibung. Sie kann beliebig viele explizite oder implizite typisierte Beziehungen zu anderen Komponenten unterhalten. Die Kanten zwischen den Komponenten erfüllen unterschiedliche Aufgaben und werden je nach Typ vom Laufzeitsystem ausgewertet. Daraus ergeben sich mehrere Dimensionen an Beziehungsgraphen, die alle mit den gleichen Komponenten arbeiten. Eine Komponente kann dabei als Knoten in unterschiedlichen Graphen zum Einsatz kommen.

Die Kontrolleinheit der Systemarchitektur ist somit ausschließlich für den Transport der Nachricht bzw. deren Kontrolle zuständig und völlig applikationsunabhängig (siehe Abbildung 3.3). Da ausschließlich alle Aufgaben innerhalb des Systems über Komponenten und deren Beziehungen definiert sind, wird der *Controller* komplett applikationsunabhängig und hat forthin drei grundsätzliche Hauptaufgaben:

- Einen Pool an Komponententypen und deren reflektive Schnittstellen bereitzustellen.

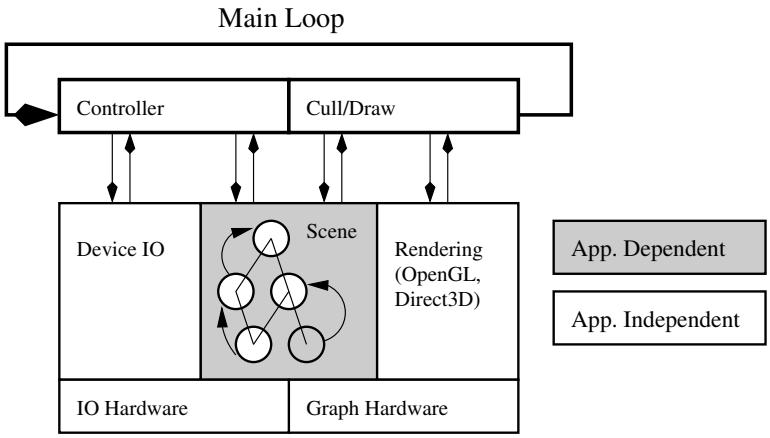

Abbildung 3.3: Systemdesign mit applikationsunabhängiger Kontrolleinheit

- Graphen mit Komponenteninstanzen und typisierten Kanten in Namensräumen zu verwalten.
- Traversierungsmethoden anzubieten, die es erlauben unterschiedliche Kantentypen auszuwerten und entsprechende Veränderungen, wie zum Beispiel Animationen, auf die Graphen anzuwenden.

Ein wichtiges Ziel dieser Architektur ist es, die Entwicklung von MR-Applikationen effizienter zu gestalten. Verglichen mit den bisherigen Abstraktionsmodellen für dynamische MR-Systeme ergeben sich diesbezüglich einige Verbesserungen, die unabhängig von den Laufzeitumgebungen und den konkreten bereitgestellten Komponenten sind.

Applikationsgerechte Abstraktion Es ist nicht mehr notwendig, dass der Anwendungsentwickler die gleichen Schnittstellen und Abstraktionen wie der Systementwickler einsetzt und voll beherrscht. Der direkte Zugriff auf Systeminterna, wie zum Beispiel Stati des Graphiksystems oder Netzwerkverbindungen, sind für den System- und Komponentenentwickler essentiell, für den Anwendungsentwickler aber weniger interessant. Weiterhin ist es nicht mehr notwendig, dass der Anwendungsentwickler die gleiche Hochsprache, in diesem Fall C++, einsetzt. Der Anwendungsentwickler kann direkt applikationsgerechte Komponenten wie Fenster, Geometrien oder Zeitgeber instanziieren und durch die Festlegung der Beziehungen die Dynamik der Anwendung modellieren.

Homogenität für Anwendungsentwickler Alle Komponenten des Systems werden einheitlich abgebildet. Eine Unterscheidung zwischen Szenenbeschreibung und Konfiguration der Laufzeitumgebung wird nicht mehr vorgenommen. Es gibt unterschiedliche Graphen für unterschiedliche Aufgaben, die aber alle dem gleichen Grundkonzept folgen.

Portabilität der Anwendung Die Anwendungen bestehen aus einer Menge an Komponentengraphen, deren Kanten zur Laufzeit traversiert und interpretiert werden. Es ist möglich, den Zustand der Komponenten und alle Verbindungen zu jedem Zeitpunkt zu realisieren, und in einer XML-Datei abzulegen. Diese Dateien können in einer anderen Laufzeitumgebung, vorausgesetzt sie stellt die gleichen Komponententypen bereit, wieder eingelesen werden. Somit sind die eigentlichen Applikationen unabhängig von der eingesetzten Hardware und Softwareplattform und somit portabel.

In den weiteren Abschnitten dieses Kapitels werden die Techniken und Datenstrukturen vorgestellt und entwickelt, die es erlauben, dieses Konzept umzusetzen.

3.3 Komponentenausprägungen und Erweiterbarkeit

Das System stellt Komponententypen in Pools bereit, wobei unterschiedliche Laufzeitumgebungen unterschiedliche Typen beinhalten können. Der Systemkern und die darauf aufsetzenden Werkzeuge müssen aus diesem Grund unabhängig von den Schnittstellen einzelner Komponenten sein. Die Komponenten und ihre Schnittstellen müssen reflektive sein, was bedeutet, dass sie zur Laufzeit über sich selbst Auskunft geben können. Weiterhin muss das System offen für zukünftige Erweiterungen sein, die immer in Form von neuen Komponenten der Kantentypen entstehen.

3.3.1 Komponentenausprägung, Identität und Status

Komponenten sind Instanzen von Komponententypen, die jeweils einem Namensraum (siehe Abschnitt 3.5.1) zugewiesen sind. Sie besitzen einen eindeutigen numerischen Identifikator, der über die Namensräume hinweg für das komplette System eindeutig ist. Die Ausprägung der Komponenten wird über den Status und die Feldwerte bestimmt.

Im Gegensatz zu Knoten in herkömmlichen Szenengraphen besitzen die Komponenten einen genau definierten Status. Dies ist notwendig, da die Komponenten

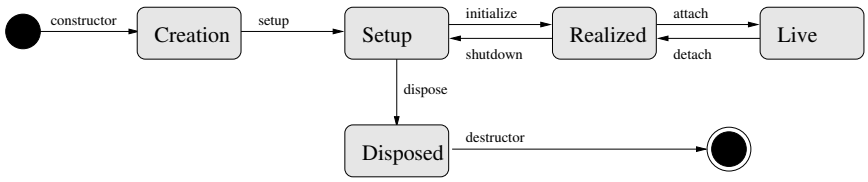

Abbildung 3.4: Zustände einer Komponente

im Gegensatz zu klassischen Szenengraphknoten abhängig von ihren Verbindungskanten Aufgaben übernehmen. Zum Beispiel sind Netzwerkverbindungen oder Datenstrukturen für die Kollisionserkennung auf- bzw. abzubauen.

Jeder Knoten besitzt dabei zu jeder Zeit einen eindeutigen Status. Der Statuswechsel (siehe Abbildung 3.4) wird dabei nicht explizit initiiert, sondern durch den Auf- bzw. Abbau von Verbindungen zu anderen Knoten eingeleitet.

Die einzelnen Stati ergeben sich dabei wie folgt:

Creation Alle inneren Zustände und Schnittstellen sind aufgebaut. Alle Felder besitzen den im Typen definierten Standardwert. Der Knoten besitzt noch keine Kanten. Es ist auch nicht möglich, den Knoten mit anderen Knoten über solche zu verbinden.

Setup Alle Feldwerte der Schnittstelle, statische und öffentliche, können verändert werden. Der Knoten besitzt keine explizite Kanten.

Realized Der Knoten wird explizit mit einem anderen Knoten verbunden. Es können weiterhin alle Werte von öffentlichen Feldern verändert werden.

Live Der Knoten hat nicht nur Kanten, sondern ist mit einem *Live*-Graphen verbunden. Ab sofort reagiert er auf Nachrichten und verarbeitet Ereignisse.

Disposed Der Knoten hat keine Verbindung zu anderen Knoten und alle weiteren Verweise und Referenzen sind gelöscht.

Der aktuelle Status kann von jeder Komponente zur jeder Zeit erfragt werden. Die einzelnen Statusübergänge rufen eine entsprechende Methode im Komponentenprozessor (siehe Abschnitt 3.3.4) auf, die vom Komponentenentwickler überladen werden kann. Somit kann jeder Knotentyp individuell auf den Statuswechsel reagieren, aber diesen nicht verhindern oder manipulieren.

Nur Knoten welche als Root Knoten in einem Kontext registriert sind (siehe Abschnitt 3.5.2) wechseln automatisch von *Creation* über *Setup* und *Realized* zu

LIVE. Sie definieren die ersten Knoten mit einem *Live* Status und dienen somit als Grundlage für alle aktiven Teilnetze.

3.3.2 Komponentenschnittstellen, Felder und Slots

Die Komponentenschnittstellen sind durch Felder definiert. Diese Felder halten zum Beispiel die Farben eines Materials, die Translation einer Transformation oder Punktpositionen einer Geometrie.

Jedes Feld wird beschrieben durch einen Namen, einen Datentypen, mögliche Ein- bzw. Ausgabe-Slots und einen für die Komponenteninstanz eindeutigen numerischen Identifikator, der zur Maskierung von Zuständen zur Laufzeit benutzt wird.

Grundsätzlich gibt es zwei unterschiedliche Feldarten:

Einzelwert-Felder (*SingleValue-Fields*) halten immer genau einen Wert von einem festen Datentypen. Sie besitzen in jedem Zeitpunkt einen gültigen Wert, auch wenn sie noch nicht beschrieben wurden.

Mehrwert-Felder (*MultiValue-Fields*) halten eine beliebige Anzahl, einschließlich Null, an Werten eines festen Datentyps.

Die einzelnen Datentypen der Felder sind unabhängig von Komponententypen und werden von dem System bereitgestellt. Ziel ist die Anzahl der Felddatentypen möglichst gering zu halten, um die Konvertierungsmatrix nicht unnötig aufzublähen. Das System stellt alle für eine MR-Applikation gängigen atomaren Basistypen bereit (siehe Tabelle 3.1).

Felder definieren zum einen die Ausprägungen einer einzelnen Knoteninstanz (zum Beispiel die Farbe eines Materials) und zum anderen die Schnittstellen zu anderen Komponenten. Die Schnittstelle einer ganzen Komponente ergibt sich dabei aus der Summe der von den einzelnen Feldern angebotenen Ein- und Ausgabe-*Slots.* Die *Slots* der Felder sind durch ihren Schnittstellentypen definiert:

Kein-Slot Das Feld kann über keine Kante mit anderen Felder verbunden werden. Der Feldwert kann nur im *Setup*-Status des Knoten beschrieben werden.

Eingabe-Slot Das Feld kann implizit oder explizit mit *Ausgabe-Slot* Feldern verbunden werden und dementsprechend Nachrichten empfangen.

Ausgabe-Slot Das Feld kann implizit oder explizit mit *Eingabe-Slot* Feldern verbunden werden und dementsprechend Nachrichten versenden.

String	UTF-8 kodierte Text Daten
Time	Zeit in Millisekunden
Bool	Binäre Daten; wahr/falsch
Float	Skalare Werte mit einfacher Genauigkeit
Double	Skalare Werte mit doppelter Genauigkeit
Int32	Integer Wert
ColorRGB	RGB kodierter Farbwert
ColorRGBA	RGB+Alpha kodierter Farbwert
Vec2F	Vektor mit zwei Komponenten und einfacher Genauigkeit
Vec3F	Vektor mit drei Komponenten und einfacher Genauigkeit
Vec4f	Vektor mit vier Komponenten und einfacher Genauigkeit
Vec2D	Vektor mit zwei Komponenten und doppelter Genauigkeit
Vec3D	Vektor mit drei Komponenten und doppelter Genauigkeit
Vec4D	Vektor mit vier Komponenten und doppelter Genauigkeit
Rotation	Rotation; Intern als Quaterion kodiert
Plane	Ebene im Raum; Intern kodiert mit Punkt und Normale
Ray	Ein Strahl; Intern kodiert mit Punkt und Richtung
Matrix4f	Matrix mit 4x4 Komponenten und einfacher Genauigkeit
Matrix4d	Matrix mit 4x4 Komponenten und doppelter Genauigkeit
Image	1D/2D/3D-Bilddaten kodiert mit RGBA Pixeln
Sound	PCM kodierte Audiodaten
NodeRef	Referenz zu einem Knoten/Komponente

Tabelle 3.1: Vom System bereitgestellte Felddatentypen

Ein/Ausgabe-Slot Das Feld kann sowohl Nachrichten empfangen als auch versenden.

Wie in der Einleitung dieses Kapitels erläutert ist die Idee der Slots nicht neu, sondern wird schon in vielen Bereichen eingesetzt. Jedoch gibt es zwei entscheidende Verbesserungen gegenüber der gebräuchlichen VRML/X3D-Spezifikation:

- Es gibt nicht nur einen Kantentypen und somit müssen die Slots sich für eine Teilmenge aller Kantentypen registrieren.
- Es ist erlaubt Slots mit unterschiedlichen Datentypen zu verbinden (z.B. Vec3f mit Float). Das System stellt eine Konvertierungsmatrix bereit, deren einzelnen Einträge zur Laufzeit parametrisiert werden können.

Zwei Felder, die einen Augabe-Slot und einen Eingabe-Slot bereitstellen, können direkt verbunden werden, vorausgesetzt der Kantentyp ist in beiden Teilmengen registriert. Dazu wird ein Kanten-Objekt generiert, das direkt zwei Felder referenziert und somit verbindet (siehe Abschnitt 3.6).

3.3.3 Statische und dynamische Felder

Die meisten Felder sind statische Felder was bedeutet, dass sie als Teil der Knotendeklaration definiert werden. Darüber hinaus muss es aber möglich sein, die Schnittstellen von Komponenteninstanzen dynamisch erweitern zu können.

Somit ist es notwendig, dass nicht nur der Knoten bzw. Komponententype eine reflektive Schnittstelle besitzt, sondern auch die einzelne Instanz Auskunft über ihre Felder und deren Eigenschaften gibt.

3.3.4 Erweiterbarkeit

Eine der wichtigsten Voraussetzungen für alle offene System ist die gernerelle Erweiterbarkeit. Um das Kernsystem und darauf aufsetzende Module unabhängig von konkreten Komponententypen gestalten zu können, werden Mechanismen eingesetzt, die es dem System erlauben Komponentenschnittstellen und deren Eigenschaften zur Laufzeit zu erfragen. Dazu ist es notwendig, Strukturen anzulegen, die wiederum Auskunft über andere Strukturen geben. Diese reflektiven Schnittstellen [84] sind die Grundlage für alle Verfahren, die einen generischen Zugriff auf Komponenten und deren Felder benötigen. Dazu zählen zum Beispiel das Anlegen der Kanten, die Traversierung der Graphen, das Lesen und Schreiben der

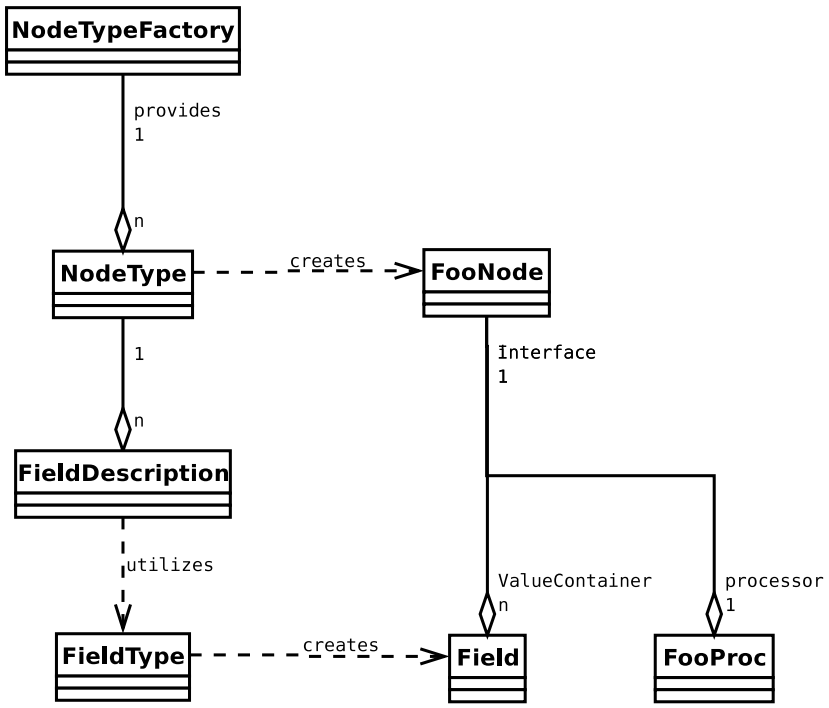

Abbildung 3.5: Strukturen zur Verwaltung der reflektiven Schnittstellen von Komponenten

Graphen, das Senden und Empfangen von impliziten Nachrichten und alle graphischen Benutzerschnittstellen.

Damit jede Komponente auch auffindbar ist wird für jeden Komponententyp eine *NodeType* Instanz in der *NodeTypeFactory* registriert (siehe Abbildung 3.5). Jedes *NodeType* Objekt beinhaltet den Komponentennamen, eine Methode zum Erzeugen von Komponenten und eine Liste von Feldbeschreibungen, die alle Attribute der statischen Felder eines Typen beinhalten. Wird die *NodeFactory* aufgefordert eine Komponente zu generieren, sucht sie sich ein passendes *NodeType* aus den internen Typenbaum und benutzt die *create* Methode, um eine Komponente zu generieren. Ist der geforderte Typ nicht bekannt, wird versucht die Komponentenbeschreibung und deren Implementierung als Prototype oder Native-Implementierung nachzuladen (siehe Abschnitt 3.4).

Die konkreten Komponenten sind mit Hilfe von zwei Klassen implementiert. Eine typspezifische Spezialisierung der *Node*-Klasse beinhaltet alle reflektiven und direkten Schnittstellenbeschreibungen und ist nur von diesen abhängig. Das Verhalten der Komponente und damit die Methoden um auf Nachrichten oder Zu-

standänderungen zu reagieren werden in einer Prozessor-Klasse implementiert. Die Prozessor-Klassen sind als Spezialisierung der *NodeProc*-Klasse deklariert. Diese Trennung hat zwei entscheidende Vorteile:

- Sollten sich die Schnittstellen ändern, muss nur die *N*ode-Spezialisierung neu erzeugt werden. Die Funktionen zur Behandlung von Nachrichten und Zustandsänderungen sind davon unabhängig.
- Für einen Knoten können die Prozessoren zur Laufzeit ausgetauscht werden, um zum Beispiel unterschiedliche Verhalten oder Implementierungen für unterschiedliche Graphikbibliotheken abzubilden.

Systemerweiterungen werden als neue Komponenten zur Verfügung gestellt. Die Komponenten können zur Laufzeit nachgeladen werden. Dabei werden die reflektiven Schnittstellen und der NodeType in der NodeFactory registriert und stehen danach der Anwendung zur Verfügung.

3.4 Deklaration und Implementierung von Komponententypen

Die Anwendung besteht aus Komponenten und deren Beziehung. Der Anwendungsentwickler instanziiert Komponenten aus dem vom System zur Verfügung gestellten Pool von Komponententypen. Darüber hinaus gibt es mehrere Möglichkeiten neue Komponententypen zu erzeugen, um entweder neue Technologien einzubinden, applikationsspezifische Typen zu entwickeln oder Komponenten durch abstrakte Typen zu klassifizieren.

3.4.1 Erweiterung der Basisknoten

Wie in dem vorangegangen Abschnitt dargelegt, besteht eine Komponente aus einer *Node* und *NodeProc* Spezialisierung. Das Node-Objekt beinhaltet nur die direkten und die generischen Schnittstellenbeschreibungen und generische Funktionen, um auf Felder oder Zustände zugreifen zu können.

Um die Erstellung der Node-Klasse zu vereinfachen, wird eine systemspezifische Schnittstellenbeschreibungssprache (NodeDescriptionFormat, NDF) ähnlich zu der Corba/IDL eingeführt. Diese Sprache erlaubt es, alle Parameter eines Komponententypen, wie zum Beispiel Vererbung und Feldbeschreibungen, in einem

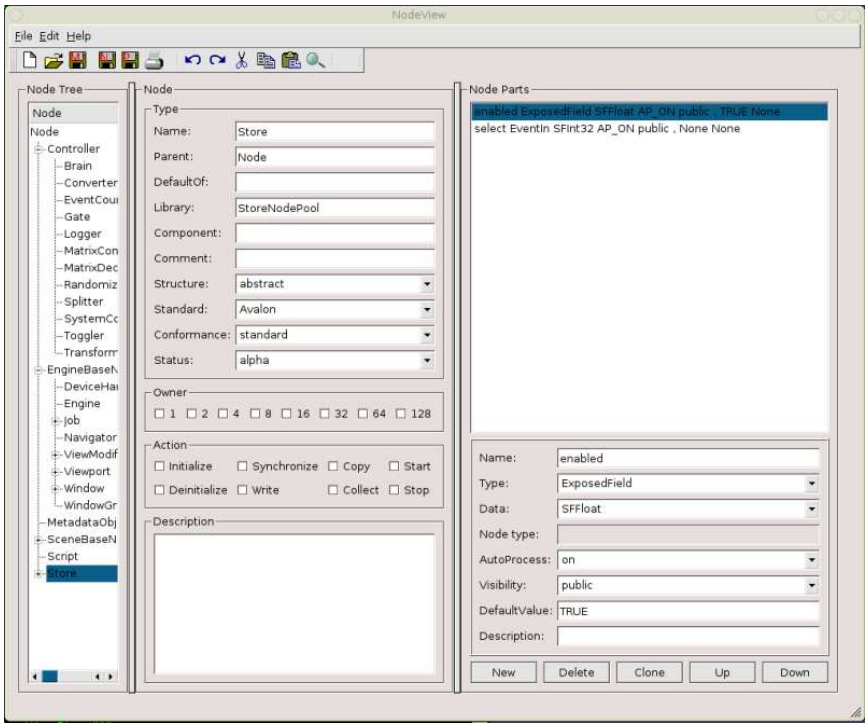

Abbildung 3.6: Graphisch-interaktives Werkzeug zu Erstellung der reflektiven Schnittstellen

XML-File zu definieren. Es wurde ein graphisch-interaktives Werkzeug erstellt, das es dem Entwickler erlaubt, die Schnittstellen interaktiv zu beschreiben (siehe Abbildung 3.6). Zur Generierung der Schnittstellen- und Funktions-Klassen wird ein Compiler bereitgestellt, der entsprechenden C++ Programmcode erzeugt. Der Entwickler muss das Skelett mit der gewünschten Funktionalität füllen und der Laufzeitumgebung zur Verfügung stellen. Dazu muss der Entwickler den Programmcode für die Zielplattform übersetzen und in Form eines nachladbaren Moduls bereitstellen. Er kann mehrere Komponenten in einem Paket zusammenfügen und das Paket komplett in der Laufzeitumgebung nachladen.

3.4.2 Implementierung durch Scrnipt-Sprachen

Zur Erstellung von Prototypen oder für die Entwicklung von applikationsspezifischen Verhalten kann es sinnvoll sein, die Funktionalität von Komponenten in eine Skriptsprache umzusetzen. Das hat den Vorteil, dass die Implementierung plattformunabhängig ist und aufwendige Übersetzungszyklen bei der Entwicklung entfal-

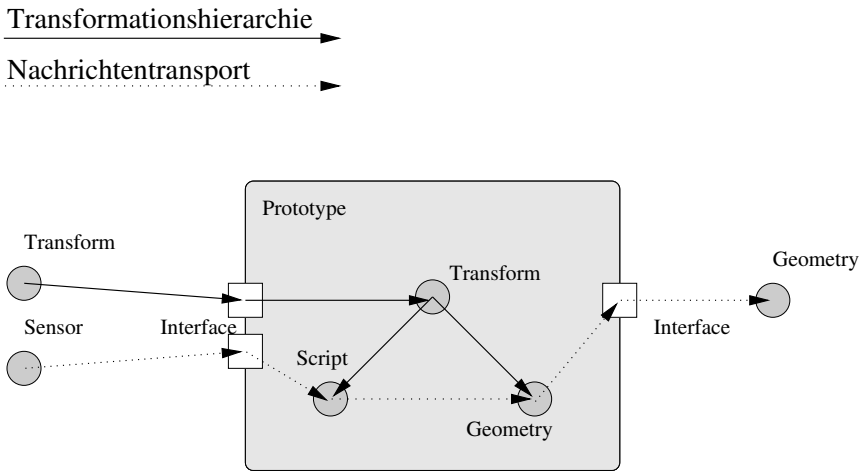

Abbildung 3.7: Deklaration von Komponenten durch Aggregation

len.

Dazu wird eine NDF Beschreibung erzeugt, die in diesem Fall keinen C++-Code generiert, sondern eine Typbeschreibung, die direkt Skriptcode zum Behandeln von Nachrichten beinhaltet. Diese Funktionalität wird dann einmalig beim Einlesen der Komponentenbeschreibung übersetzt und dem System zur Verfügung gestellt.

Komponenten haben Zugriff auf alle Nachrichten und deren Inhalt, können aber nicht auf die untersten Systemschnittstellen, wie zum Beispiel den Darstellungsgraphen, zugreifen. Aus diesem Grund und aus dem schlechteren Laufzeitverhalten eignen sich Scriptimplementierungen vor allem zum Generierung von applikationsspezifischen Komponenten, die technologisch weniger anspruchsvoll sind.

3.4.3 Aggregation

Um die Wiederverwendung von Teilgraphen zu vereinfachen, können diese zu einer neuen Komponente zusammengefasst werden. Dabei wird ein neuer Komponententyp spezifiziert, registriert und der Teilgraph als Prototyp abgelegt. Die eingeschlossenen Schnittstellen für Kanten können teilweise nach außen gegeben werden und definieren somit die Schnittstellen der neuen Komponente (siehe Abbildung 3.7). Zur Instanziierung wird der Prototypen-Teilgraph in einen eigenen Namensraum kopiert und als eigenständige Komponente zur Laufzeit behandelt.

3.5 Namensräume und Laufzeit-Kontext

Für den Aufbau von nicht-trivialen Szenen und Applikationen ist es wünschenswert, dass der Applikationsentwickler explizit und implizit durch Prototypen Komponenteninstanzen gegeneinander abgrenzen kann. Wenn zum Beispiel eine Szene mehrere Modelle von unterschiedlichen Häusern beinhaltet, sollte das System in der Lage sein für jedes Haus eine Komponente mit dem Namen *Dach* zu verwalten.

Zu diesem Zweck werden an dieser Stelle Namensräume für Komponenten eingeführt, die ähnlich zu den Namenräumen in modernen Programmiersprachen (zum Beispiel C++) funktionieren. Der Unterschied ist, dass in Programmiersprachen Namensräume Deklarationen abgrenzen und in diesem Fall Instanzen in Räume aufgeteilt werden.

Grundsätzlich sind alle Namensräume unabhängig von den darin registrierten Komponenten. Jedoch gibt es zwei spezielle Namensräume, die direkt an den Laufzeit-Kontext gebunden sind und die ersten Namensräume mit aktiven Knoten definieren.

Die Ausprägungen der Namensräume und das Zusammenspiel mit dem Laufzeit-Kontext wird in den folgenden Abschnitten näher erläutert.

3.5.1 Hierarchie von typisierten Namensräumen

Ein Namensraum beinhaltet einen Teilgraphen bestehend aus Komponenteninstanzen und Kanten. Darüber hinaus besitzt er selbst wieder einen Namen und eine Liste von Komponententypen. Die Liste der Komponententypen bestimmt die Art der Komponenten, welche in diesem Namensraum registriert werden können. Jede Komponente ist zu einem Zeitpunkt bei genau einem Namensraum registriert, wobei es möglich ist, dass die Komponente den Namensraum zur Laufzeit wechselt.

Um Namensräume adressieren zu können, sind sie im Allgemeinen als Kind eines bereits bestehenden Namensraumes definiert. Da dies rekursiv geschehen kann, entsteht eine Hierarchie von Namensräumen. In Abbildung 3.8 existieren Insgesammt drei Komponenten mit dem Namen *A*. Da sie in drei unterschiedlichen Namensräumen registiert sind ist es eine gültige Benamung.

Weiterhin gibt es spezielle Namensräume die keinen Vater besitzen, sondern direkt mit dem Laufzeit-Kontext verbunden sind. Die Aufgabe dieser speziellen Namensräume wird in den weiteren Abschnitten genauer dargelegt.

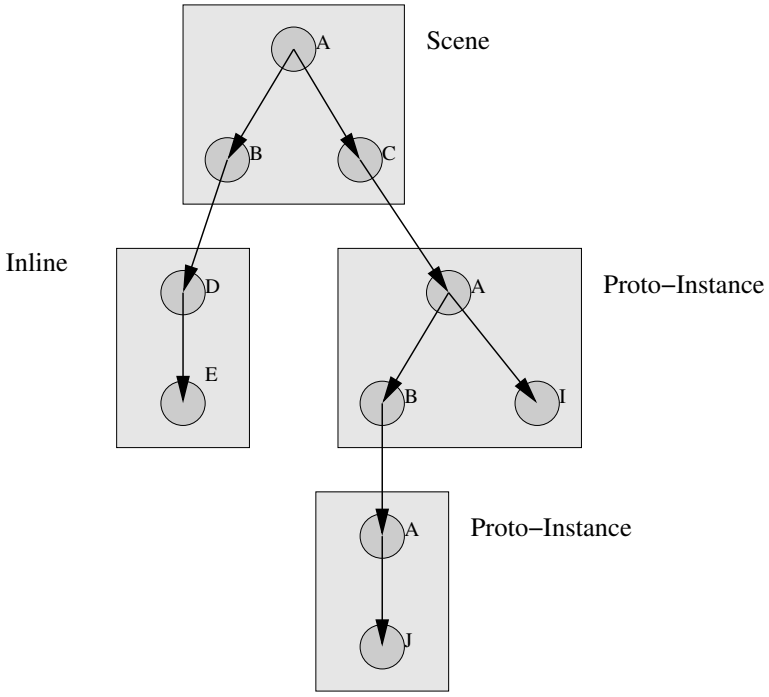

Abbildung 3.8: Gülige Hierarchie von Namensräumen

3.5.1.1 Szene

Der Szenen-Namensraum definiert die Wurzel für alle Namensräume die Komponenten beinhalten, welche die aktive virtuelle Welt beschreiben. Sie beinhalten ausschließlich Komponenten die von Typen *SceneBaseNode* abgeleitet sind. Um den Einsatz von Modellierungssystemen wie 3DMax [8] und Maya [3] zu vereinfachen, wird die X3D-Spezifikation [143] als Grundlage für alle in der Szene erlaubten Knoten benutzt. Die X3D-Spezifikation unterstützt bereits einen Szenengraphen und Feldverbindungen zum Austausch von Nachrichten zwischen Komponenten. Aus diesem Grund ist der Standard hervorragend als Grundlage geeignet. Die wichtigsten Typen lassen sich in die folgenden Gruppen einteilen:

Timer Komponenten, die auf zeitliche Veränderungen reagieren und entsprechend ihrer Konfiguration Ausgangssignale erzeugen.

Sensoren Knoten, die auf Benutzerinteraktionen oder Veränderungen im Graphen, zum Beispiel die Kollision von zwei Objekten, reagieren und entsprechend ihrer Ausprägung Nachrichten verschicken.

Logikbausteine Simple Logikkomponenten, wie zum Beispiel FlipFlops und Schranken zur Steuerung der Applikation.

Interpolatoren Funktionskomponenten, die mit Hilfe von linear definierten Teilfunktionen einen Eingabewert in einen Funktionswert überführen.

Visuelle Komponenten Traditionelle graphische Objekte, wie Geometrien und Materialien.

Akustische Komponenten Objekte zur Beschreibung von dreidimensionalen akustischen Welten.

Gruppen Komponenten, die andere Instanzen gruppieren. Dazu gehören auch Transformationsknoten, welche Teilgraphen in der visuellen oder akustischen Welt transformieren.

Script Definieren Script-Knoten, deren Verhalten nicht in C++ sondern einer Scriptsprache, wie zum Beispiel ECMAScript [36] oder Java [48], definiert sind.

Inline Definieren einen Kind-Namespace, dessen Knoten aus einem als URL mitgegebenen Stream aufgebaut werden.

Durch die Inlines ist es möglich, rekursiv eine beliebige Anzahl von Kindernamensräumen in der Szene zu definieren. Diese können immer direkt über die Szene adressiert werden.

Es können beliebig viele Szene-Namensräume bestehen, jedoch gibt es immer nur einen aktive Scene pro Laufzeit-Kontext.

3.5.1.2 Engine

Grundsätzlich ist die Szene eine interaktive, akustische und visuelle Welt, die unabhängig von den Ein- und Ausgabeumgebungen, zum Beispiel Cave [30] oder Desktop, ist. Die Sensoren sind alle unabhängig von den entsprechenden Eingabegeräten und somit umgebungsunabhängig (siehe Kapitel 4).

Der Engine Namespace beinhaltet alle Komponenten, die zur applikationsunabhängigen Beschreibung und Konfiguration der Laufzeitumgebung benötigt werden. Das sind unter anderem Knoten, welche sequenzielle und parallele Abläufe, Geräte und Geräteklassen oder Fenster und Cluster spezifizieren.

Die wichtigsten Typen lassen sich in die folgenden Gruppen aufteilen:

Jobs Definieren Teilaufgaben, die in einem Durchlauf entweder sequenziell oder parallel abgearbeitet werden. Dazu gehören das Einlesen von Eingabegeräten, das Bereitstellen von Netzwerkschnittstellen oder das Rendering selbst.

DeviceHandler Definieren das Mapping für externe Geräteabstaktion (siehe auch Kapitel 4) auf die Label der internen Sensoren.

Windows Definieren Lokale oder Cluster-Windows.

Viewports Definieren Ausschnitte innerhalb eines einzelnen Windows, die für die Darstellung der Szenen genutzt werden.

CameraModifier Definiert Modifikationen auf der aktuellen Kamera bzw. Viewpoints. Diese Modifikationen erlauben es zum Beispiel unterschiedliche Stereo oder Tile-View-Parameter zu setzen.

Alle Abläufe innerhalb der Engine lassen sich über die Kanten zwischen den Komponenten definieren. Zum Beispiel steuern Nachrichten die sequenzielle und parallele Abarbeitung von Jobs.

Es können beliebig viele Engine-Namensräume bestehen, jedoch gibt es immer nur eine Engine aktive pro Laufzeitkontext.

3.5.1.3 Prototypen-Deklarationen und Instanzen

Prototypen Deklarationen beinhalten Namensräume mit Teilgraphen, die durch Aggregation einen neuen Typen beschreiben. Wird dieser Prototyp instanziiert, wird ein weiterer Namensraum erzeugt und die Komponenten der Deklaration kopiert. Dies kann rekursiv geschehen, womit eine neue Hierarchie an Namensräumen entsteht.

3.5.2 Laufzeit-Kontext

Für Laufzeitumgebungen ist es wünschenswert, dass sie genau einen Zugangspunkt für alle globalen Einstellungen, wie zum Beispiel die aktuellen Auflösungswert (siehe Kapitel 5) für die skalierbaren Darstellungsverfahren, und Verweise auf alle aktiven Namensräume hat.

Zu diesem Zweck stellt das System einen Laufzeit-Kontext bereit. Er verweist immer genau auf einen aktiven Engine-Namensraum und eine aktive Szene. Der Laufzeit-Kontext wird von der Laufzeitumgebung generiert und verwaltet.

3.6 Typisierte Kanten und Nachrichtenverarbeitung

Als Grundlage für den in Abschnitt 3.2 vorgestellten mehrdimensionalen Graphen dienen die typisierten Kanten. Diese Kanten werden primär genutzt, um Beziehungen und Nachrichtenkanäle zwischen Knoten zu definieren. An dieser Stelle sollen die Ausprägungen der Kanten und die Nachrichtenverarbeitung weiter diskutiert werden.

3.6.1 Typisierte Kanten

Alle Kanten besitzen einen Typ, der die Knotenbeziehung definiert und die Funktion der Kante bestimmt. Sie verbinden immer zwei Knoten beziehungsweise deren Felder. Grundsätzlich werden explizite und implizite Kanten unterschieden:

Explizite Kanten Explizite Kanten werden entweder vom Benutzer oder vom System für interne Aufgaben angelegt. Sie verbinden genau zwei Felder bzw. Slots und bestehen, bis sie explizit gelöscht werden.

Implizite Kanten Implizite Kanten werden ausschließlich vom System angelegt und wieder gelöscht. Die Verbindungen sind nicht permanent, sondern nur für einen einzigen Durchlauf gültig. Nur solange sind die Komponenten verknüpft. Dadurch ist es zum Beispiel möglich, Daten an alle Komponenten zu schicken, die einen geometrischen Abstand nicht überschreiten.

3.6.2 Nachrichtenverarbeitung und Synchronisation

Wird über eine Kante eine Nachricht an eine Komponente geschickt, so definiert die Verhaltensbeschreibung der Implementierung, sei sie in C++ oder einer Scriptsprache, die Reaktion. Im Allgemeinen prüft bzw. konvertiert und verarbeitet die Komponente die Nachricht. Dabei ist es möglich, dass sie direkt ihren inneren Zustand ändert und beliebig viele Nachrichten über andere Kanten verschickt. Darüber hinaus gibt es die Möglichkeit, einmal pro Frame auf alle Veränderungen zu reagieren. Zu diesem Zweck werden zwei neue Methoden im Knotenprozessor bereitgestellt, welche bei Bedarf zu überladen sind.

fieldChanged Wird bei jeder Nachricht oder Feldänderungen in den aktiven Knoten aufgerufen. Aufgabe der Methode ist es, direkt auf die Nachricht zu reagieren.

nodeChanged Wird einmal pro Durchlauf für alle Komponenten aufgerufen, die mindestens eine Nachricht und oder eine Feldänderung im aktuellen Durchlauf erfahren haben.

3.7 Traversierung der Graphen

Alle Systeme, die einen Szenegraph zur Darstellung oder Manipulation von virtuellen Welten einsetzen, stellen Funktionen und Objekte zur Traversierung des Graphen bereit. Dabei durchlaufen diese Objekte die Knotenhierarchie rekursiv, um unterschiedliche Aufgaben auszuführen. Es gibt in der Literatur unterschiedliche Namen für diese Objekte, der gebräuchlichste ist *Action*.

In den bisherigen Systemen wurden die Action-Objekte nur auf die Szenenhierarchie eingesetzt. Durch die in dieser Arbeit vorgestellte Multi-Graphen Architektur, bestehend aus Komponenten und unterschiedlichen Kantentypen, muss das bisherige Konzept erweitert und angepasst werden. Zu diesem Zweck werden die Action-Objekte so definiert, dass die Instanzen über eine Liste von Kantentypen parametrisiert werden können. Mit Hilfe dieser Liste ist es nun möglich die Action nur auf einem Einzelnen, einer Auswahl oder auf allen Kantentypen arbeiten zu lassen.

Grundsätzlich benutzten die Objekte beim Traversieren Methoden der Klassenprozessoren (*NodeProc*-Klassen, Abschnitt 3.3.4). Dabei werden für jede Action unterschiedliche Methoden aufgerufen, die in den einzelnen Klassen separat überladen werden können. Aufgabe dieser Methoden ist es, anhand der Action-Parameter die nächste Kante auszuwählen und an die Action weiterzugeben. Zusätzlich können beliebige weitere Arbeitsschritte vor oder nach der weiteren Traversierung platziert sein. Durch den Austausch der Klassenprozessoren ist es möglich, dieses Verhalten zur Laufzeit anzupassen.

Für Rendering-Systeme, wie OpenSG [65] oder Performer [111], ist das wichtigste bei der Traversierung sicherlich die Synthetisierung von Bildern. Dabei traversiert eine Render-Action den Baum und steuert, neben der Auswahl an sichtbaren Objekten und dem Minimieren der Zustandswechsel, die Zustände und Parameter der Graphikhardware. Das hier vorgestellte Rahmensystem steuert die Graphikhardware nicht direkt, sondern synchronisiert für diese Aufgabe einen speziellen, für den Anwendungsentwickler unsichtbaren OpenSG-Graphen. Die Actions sind vorwiegend Funktionen, die der System- und Applikationsentwickler frei einsetzen kann, um den Graphen zu untersuchen oder zu manipulieren:

CollectAction Traversiert den Graphen und sammelt alle Knoten deren Typen mit einer in der Action spezifizierten Liste übereinstimmen.

CopyAction Kopiert einen Graphen mit allen Komponenten und Kanten

NameSpaceTransferAction Traversiert den Graphen und überführt die besuchten Knoten von einem Namensraum in einen anderen.

TransformAction Ermöglicht unterschiedliche Transformationen und Optimierungen auf den besuchten Knoten zu initiieren.

SerializeAction Ist die Basisklasse für unterschiedliche Actions, die es erlauben den Graphen zu serialisieren, um ihn in einer Datei zu speichern oder über das Netzwerk zu übertragen.

Alle Actions können so konfiguriert werden, dass sie über Namensräume hinweg arbeiten und den Komponenten-*State* (siehe Abschnitt 3.3.1) berücksichtigen.

3.8 Netzwerktransparente Schnittstellen

Das bisher vorgestellte Applikationsmodell geht davon aus, dass die komplette Anwendung innerhalb des Systems unter dessen Kontrolle abläuft. Für eine Vielzahl von Applikationen ist es aber notwendig, dass sie Teile des Graphen oder der Komponentenparameter durch externe Schnittstellen manipulieren können. Systemübergreifende Schnittstellen werden zum Beispiel zum Anschluss von externen graphischen Benutzerschnittstellen benötigt oder bei Simulationssystemen, die das System als Visualisierungssystem einsetzen.

Diese externen Schnittstellen werden als netzwerktransparenter *Application-Service* (Siehe Abbildung 3.9) vom System bereitgestellt. Sie setzen einen ZeroConf-kompatiblen [133] Mechanismus ein, um ihre Dienstleistungen netzwerktransparent anzubieten. Daneben existieren Werkzeuge welche die ZeroConf Nachrichten und somit die Diestleistungen selbst über einen speziellen Web-Server zu verfügung stellen (siehe Abbildung 3.10). Diese dynamisch generierten HTTP-Seiten erlauben es dem Anwender Dienstleistung auszuwählen und zu überwachen.

Prinzipiell bieten alle Dienstleistungen die gleichen Funktionen an:

- Erzeugen/Löschen von Komponenten
- Erzeugen/Löschen von Kanten

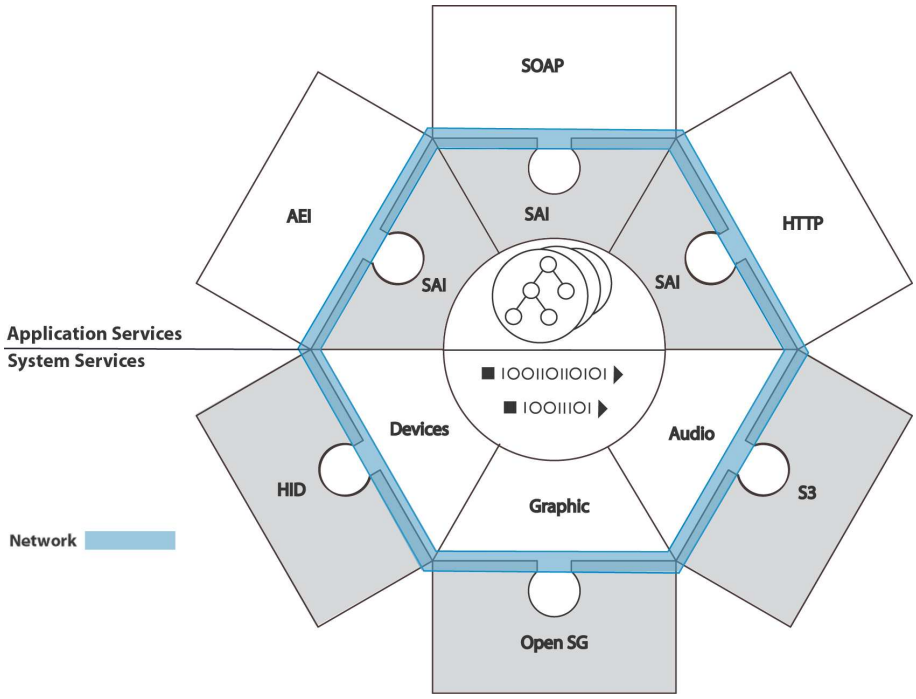

Abbildung 3.9: Application und System-Services

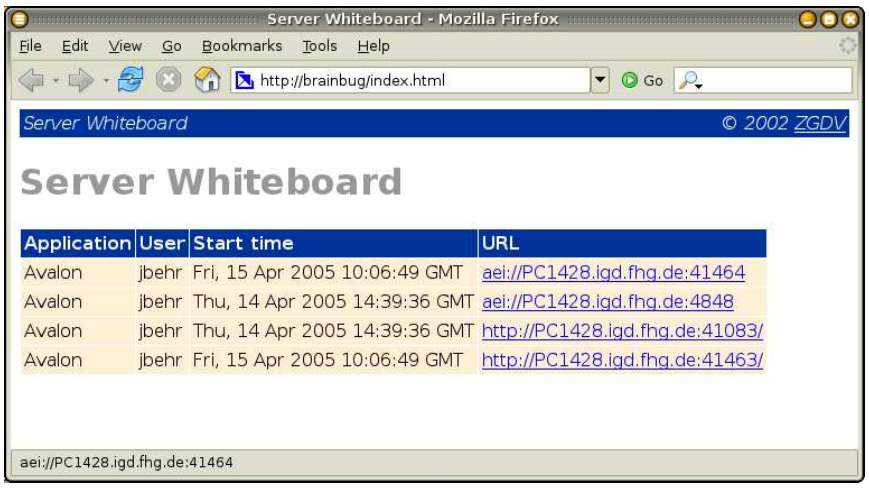

Abbildung 3.10: Dynamische Liste aller Verfügbaren *Service*-Schnittstellen

- Erzeugen/Löschen von KomponentenTypen
- Senden/Empfangen von Nachrichten
- Setzen/Lesen von Feldern

Die Schnittstellen der Funktionen sind immer applikationsunabhängig, jedoch erlauben sie den applikationsabhängigen Graphen zu manipulieren. Das System stellt für unterschiedliche Anforderungen drei Service-Netzwerkschnittstellen bereit, die in den weiteren Abschnitten genauer erläutert werden. Darüber hinaus benutzt das System noch weitere Netzwerkschnittstellen, um zum Beispiel die grafische Darstellung und die Ausgabe von Sound zu verteilen, und allgemein Ein-/Ausgaberäte netzwerktransparent anzusprechen. Diese Schnittstellen sind aber an diesen Stellen aus zwei Gründen nicht interessant: Zum einen sind sie für den Applikationsentwickler nicht sichtbar und auch nicht manipulierbar, und zum anderen haben diese Schnittstellen immer nur eine kleine begrenzende Sicht auf das System und sind für ganz spezielle Aufgaben optimiert.

3.8.1 HTTP-Service

Um die Entwicklungszeiten gering zu halten und um laufende Applikationen auch über das Netzwerk möglichst flexibel untersuchen und manipulieren zu können, integriert das System einen HTTP-Server. Dieser Web-Server erlaubt es dem Anwendungsentwickler den aktuellen Graphen interaktiv, vergleichbar mit Konfigurationswerkzeugen von Netzwerkservern, anzusprechen.

Es gibt nur wenige bisherige Systeme, die einen Web-Server zur Steuerung von VR/AR-Applikationen einsetzen [83, 100]. Diese Systeme stellen jedoch keinen Web-Server als integralen Teil bereit, sondern versuchen über bestehende Server und CGI-Scripts Schnittstellen zu exportieren.

An dem hier vorgestellten Ansatz ist neu, dass Designer und Entwickler von einer beliebigen Maschine im Netz mit einem gebräuchlichen Web-Browser die Graphen einer laufenden Applikation untersuchen und manipulieren können (siehe Abbildung 3.11).

Darüber hinaus erlaubt der Server die oben aufgeführten Service-Funktionen, direkt in HTTP-GET zu kodieren. Zum Beispiel wird das Kommando zum Setzen eines Feldes wie folgt in HTTP abgebildet.

```
http://localhost/setValue?node=scene::ball&
        field=set_visible&value=TRUE
```

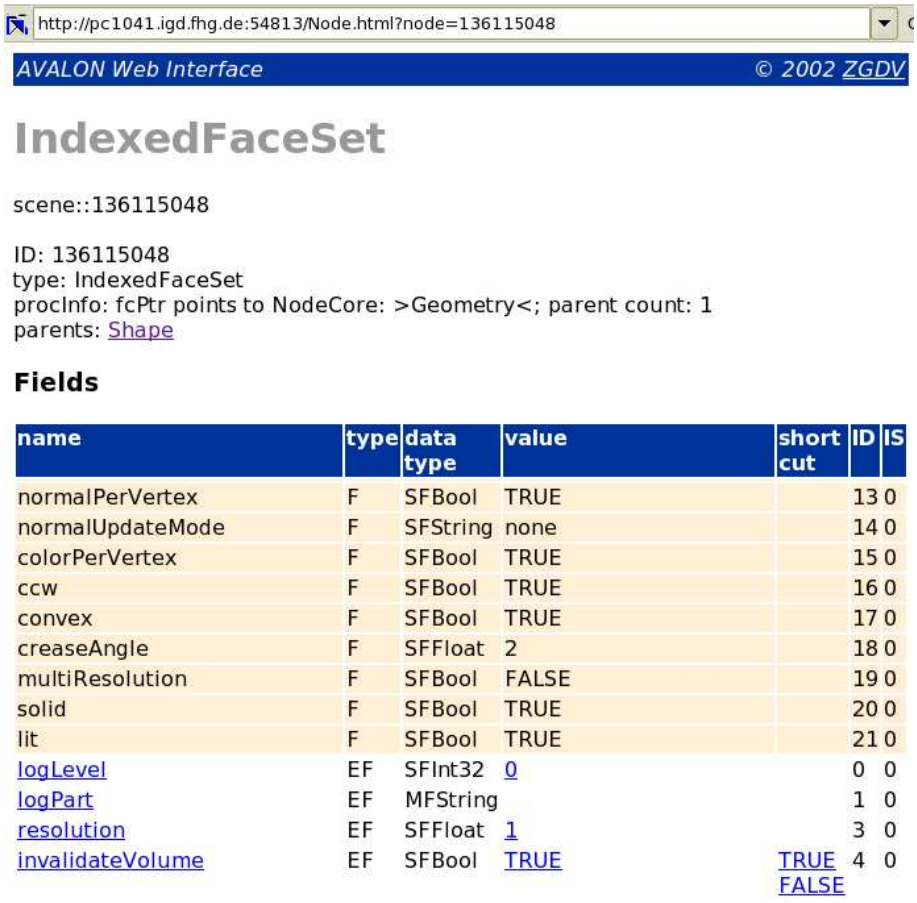

Abbildung 3.11: Graphisch-Interaktive HTTP Schnittstelle eines Knoten

Das Interessante ist, dass fast alle Programme und Anwendungen die die Anbindung bzw. die Ausführung eines *Web-Link* anbieten, automatisch mit dem VR-System kommunizieren können. Somit ist es zum Beispiel möglich mit einer Powerpoint-Präsentation, einem PDF-Dokument oder einem interaktiven Flash-Film VR/AR-Anwendungen zu steuern.

3.8.2 SOAP Service

In den letzten Jahren hat eine neue Netzwerktechnologie in vielen Bereichen Einzug gehalten: *WebServices* oder *SOAP* haben viel Aufmerksamkeit auf sich gezogen. Ein Grund hierfür ist sicherlich, dass SOAP als Basis für alle Netzwerknachrichten in dem Microsoft .NET Framework und in Sun's OpenNetworkEnvironment (ONE) dient. SOAP ist ein Basisprotokoll für den Austausch von Informationen in verteilten Umgebungen. Es ist ein Standard zum Kodieren von Nachrichten in XML und zum Transportieren über HTTP.

SOAP als W3C-Recommendation [140] wird von einer Vielzahl von Programmiersprachen direkt unterstützt, oder es gibt zumindest etablierte Implementierungen. Die Microsoft .NET und Java Standards spezifizieren direkt plattformunabhängige SOAP-Interfaces. Für PHP, Python und C++ gibt es eine Vielzahl an freien und kommerziellen Implementierungen auf dem Markt.

Somit können Entwickler diese Sprachen direkt einsetzen und das Rahmensystem ansteuern, ohne spezielle VR/AR-spezifische Schnittstellen einbinden zu müssen. Zusätzliche Schnittstellenklassen sind nicht notwendig.

Der folgende Code zeigt zum Beispiel einen Abschnitt Python-Code, der einen Feldwert setzt:

```
...
import sys
import SOAP
...
ball = SOAP.SOAPProxy ('http://localhost/SOAP',
       namespace='scene::ball');
...
ball.setField(name=translation,value='1 0 0');
...
```

Der *namespace* in diesem Beispiel deklariert den XML-Namespace und hat nichts mit den weiter oben erläuterten Komponentennamensräumen zu tun.

Interessant an dieser Lösung ist vor allem, dass nur ein systeminterner HTTP-Server sowohl die SOAP als auch die HTTP-Service-Nachrichten (siehe Abschnitt 3.8.1) verarbeiten kann was die Integration und Implementierung vereinfacht.

3.8.3 External Scene Interface Service

Dieser Service bietet die gleichen Funktionen an, jedoch benutzt es kein Standardprotokoll, sondern schickt systemspezifisch kodierte Pakete über TCP/IP. Das hat den Nachteil, dass der Anwendungsentwickler auf Client und Server-Seite spezielle Java/C++-Bibliotheken einsetzen muss. Auf der anderen Seite stehen zwei entscheidende Vorteile:

Datendurchsatz Vor allem bei relativ großen Datenmengen, wie Bildern oder Geometrien, sind text-kodierte Blöcke aufwendig zu übertragen.

Asynchron Das HTTP-Protokoll unterstützt nur die synchrone Kommunikation. Der Client ruft eine Funktion (zum Beispiel setField()) im Server auf, jedoch kann der Server nicht unabhängig Nachrichten an den Client senden. Das ist aber sinnvoll, wenn man Veränderungen des Graphen überwachen will (zum Beispiel auf eine Kollision reagieren).

3.9 Laufzeitumgebungen

Die Laufzeitumgebungen des Rahmensystems stellen Softwareplattform zur Ausführung von VR/AR Applikationen bereit. Jede Umgebung verwaltet im Kern den Laufzeitkontext, Komponententypen und die aktiven Graphen. Dabei sind verschiedene Umgebungen auf unterschiedlichen Hardware- und Betriebsystemplattformen implementiert. Die unterschiedlichen Laufzeitumgebungen beinhalten keine anwendungsspezifischen Komponenten, sondern stellen nur unterschiedliche System- (API) und Benutzerschnittstellen (GUI) bereit. Wie am Anfang dieses Kapitels beschrieben, besteht die Anwendungsentwicklung darin, vorhandene Komponenten zu Graphen zusammenzufügen, und die Beziehungen zwischen ihnen festzulegen. Dies geschieht mittels einer XML-Beschreibung, die beim Start der Anwendung von der Laufzeitumgebung eingelesen wird. Somit ist die eigentliche Applikation immer portabel, da sie nur aus XML-Dateien besteht, welche von

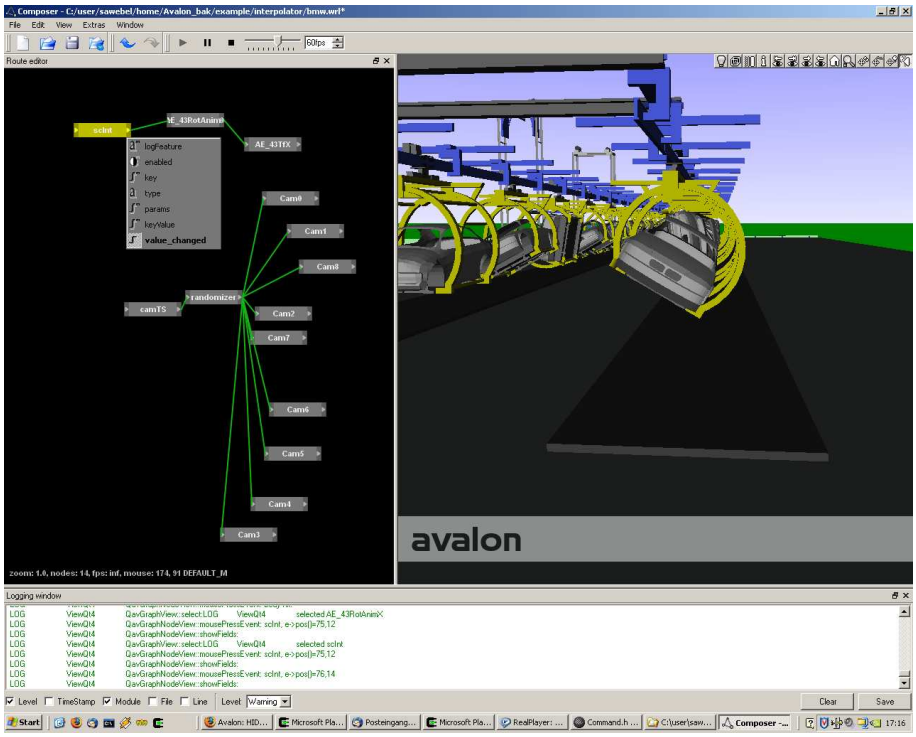

Abbildung 3.12: Graphisch-Interaktive Laufzeitumgebung zum Editieren und Überwachen von Anwendungen

den Laufzeitumgebungen interpretiert werden. Diese XML-Beschreibung kann mit jedem beliebigen Text-Editor bzw. XML-Editor erzeugt werden, was jedoch naturgemäß eine relativ unkomfortable Lösung darstellt. Um die Anwendungsentwicklung zu erleichtern, wurden eine Reihe von Werkzeugen entwickelt, die den Applikationsentwickler unterstützen.

Ein wichtiges Werkzeug ist ein Laufzeitsystem, die es dem Entwickler erlaubt, den aktuellen Zustand des Graphen graphisch darstellt und zu manipulieren. Die graphische Darstellung zeigt alle Komponenten, ihre aktuellen Parameter, und die Verknüpfungen, die zwischen den Komponenten existieren. Der Graph kann während der Laufzeit manipuliert werden, d.h. es können Komponenten hinzugefügt oder entfernt, Parameter modifiziert und Verknüpfungen erzeugt oder gelöscht werden (siehe Abbildung 3.12). Weiterhin ist es möglich, logische Breakpoints und Ereigniszähler für Nachrichten anzulegen, um die Fehlersuche zu erleichtern.

3.10 Zusammenfassung

In diesem Kapitel wurde ein einheitliches Modell von Graphen entwickelt, das es erlaubt, alle dynamischen Aspekte einer MR-Applikation mit Hilfe von Komponenten und typisierten Kanten zu modellieren. Das Konzept abstrahiert die klassischen topologischen Beziehungen, sowie statische und dynamische Nachrichtenkanäle als Kanten innerhalb unterschiedlich gerichteter Graphen. Daraus abgeleitete Anforderungen wurden diskutiert und entsprechende Lösungen entwickelt. Dabei wurden insbesondere Problemstellungen wie Erweiterbarkeit, die Verwaltung und Bereitstellung von statischen und dynamischen Metabeschreibungen und die Verarbeitung von Nachrichten untersucht und neuartige Ansätze entwickelt.

Durch das einheitliche Modell wird der Entwicklungsprozess zur Erstellung von MR-Anwendungen gegenüber dem Stand der Technik wesentlich vereinfacht. Das hier vorgestellte Systemdesign vereinfacht die Umsetzung der zu Beginn gestellten Anforderungen an ein modernes MR-System nicht automatisch - aber es ist die Grundlage für weitere Verbesserungen.

Anwendungslogik und Dynamik Das dynamische *verdrahten* von Komponenten und der Nachrichtenaustausch über Beziehungskanten erlaubt es, die Anwendungslogik und -dynamik direkt im Modell zu beschreiben. Alle Veränderungen der Szene und der Knotenattribute können vom Anwendungsentwickler in einem einheitlichen Modell von Beziehungen beschrieben werden. Simulatoren, die spezielle Teilaufgaben übernehmen, erleichtern darüber hinaus die Applikationsentwicklung.

Sensorik und Interaktion Datenströme von beliebig vielen realen Objekten werden mit Hilfe der Sensoren in das System überführt. Dabei werden nicht nur Video- und Audioströme unterstützt, sondern auch mehrdimensionale Messströme für Translationen und Rotationen. Da jedes Ereignis, und somit auch alle Datenströme von externen Sensoren mit Zeitstempeln versehen sind, können sie innerhalb der Applikationen mit einfachen Logikbausteinen gemischt und synchronisiert werden. Dies wird in Kapitel 4 noch genauer untersucht.

Skalierbarkeit der Laufzeitumgebung Die Applikationsbeschreibung besteht aus nur einer oder mehreren ASCII-/XML-Dateien, die Komponenteninstanzen und ihre Beziehungen definieren. Diese können auf jeder Plattform, auf der die Laufzeitumgebung bereitsteht, zur Ausführung gebracht werden. Der

Entwickler muss nur sicherstellen, dass die Komponenten als systemspezifische Implementierung bereitstehen, um sie bei Bedarf nachzuladen. Darüber hinaus ist das System in der Lage, einzelne isolierte Ereignisflussgraphen automatisch zu extrahieren und parallel auszuführen, um die Anwendung auf mehrere Prozessoren zu verteilen. Dieser Aspekt wird in Kapitel 5 noch genauer untersucht.

Die neuartige Systemarchitektur und -Implementierung erlaubt es dem Anwendungsentwickler, schnell und mit nur geringem Aufwand Mixed-Reality-Applikationen zu erstellen, die portabel und skalierbar sind, und dennoch die vorhandene Hardware-Plattform in der bestmöglichen Weise ausnutzen.

Kapitel 4

Flexible Interaktion und Sensorik

Interaktion ist eine Schlüsseltechnologie für MR-Anwendungen und Systeme. Ohne Benutzterinteraktion sind VR- und AR-Applikationen bestenfalls Stereofilme, die in Echtzeit berechnet werden. Der Umstand, dass der Benutzer Standpunkt, Blickrichtung und/oder Teile der simulierten Welt direkt oder indirekt verändern kann ermöglicht ihm, in die Welt aktiv einzutauchen und nicht nur passiv zu konsumieren.

Interaktionen innerhalb MR-Systemen unterscheiden sich in zwei Bereichen von gewöhnlichen Desktop-Anwendungen.

- Herkömmliche interaktive 2D-Desktop-Anwendungen bedienen sich im Allgemeinen nur weniger Ein- und Ausgabekanäle. Der Benutzer generiert Eingaben mit Maus und Tastatur und bekommt Ausgaben über das Window-System visuell und in Ausnahmefällen akustisch präsentiert. VR-Umgebungen hingegen müssen eine Vielzahl verschiedener Ein- und Ausgabekanäle mit unterschiedlichen Frequenzen bedienen.

- Die meisten 2D-Desktop-Anwendungen unterstützen das standardisierte Interaktionsmodell der entsprechenden Umgebung. Diese Interaktionsmodelle definieren graphische Elemente mit Knöpfen und Auswahllisten und deren Verhalten. Sie sind für die jeweilige Desktop-Umgebung festgelegt [89][6][128][16]. In der Welt der VR-Systeme konnten sich vergleichbare Standards nicht etablieren. Fast alle VR-Systeme bieten ebenfalls Interaktionsbausteine an, jedoch auf unterschiedlichen Abstraktionsebenen und mit inkompatiblen Modellen.

Die Schaffung eines einheitlichen Standards von graphisch-interaktiven Elementen

für MR-Systeme ist für viele Anwendungen kein ausreichender Lösungsansatz. Die Einbindung und Verarbeitung von Datenströmen, wie zum Beispiel Videodaten und die direkte Manipulation von Teilen in der Welt, kann nicht hinreichend durch vordefinierte graphisch-interaktive Elemente geleistet werden.

Aus diesem Grund beschränken sich typische VR-Umgebungen je nach ihrem Einsatzschwerpunkt auf ein Abstraktionsmodell, und überlassen es dem Anwendungsentwickler geeignete Interaktionsformen abzuleiten. VR-Umgebungen die möglichst offen und flexibel unterschiedliche Applikationsgruppen unterstützen wollen, bieten daher oft nur eine minimale Abstraktion für Ein- und Ausgabegeräte. Dadurch ist die Entwicklung von konkreten Anwendungen entsprechend aufwendig.

In dieser Arbeit wird ein neuartiges Modell vorgestellt. Dabei wird keine einzelne Abstraktion für Geräte oder Methoden entwickelt, sondern ein dreistufiges Sensor-Konzept vorgestellt, das dem Entwickler eine flexible aber auch zugleich mächtige Umgebung zur Verfügung stellt:

Sensoren für Datenströme *Data Stream Sensor (DSS)* Objekte erlauben die direkte Anbindung von Ein- und Ausgabeströmen unabhängig von einzelnen Geräten oder Interaktionsaufgaben.

Sensoren zur direkten Manipulation *Direct Manipulation Sensor (DMS)* Objekte erlauben dem Anwender Teile der Szene grafisch-interaktiv und direkt zu manipulieren. Diese Sensoren reagieren auf Veränderungen des Benutzermodells (zum Beispiel der Position der Hand), sind selbst aber unabhängig von konkreten Eingabegeräten. Die Veränderung des Benutzermodells wird direkt über *DSS* Objekte gesteuert.

Sensoren zur indirekten Manipulation *Indirect Manipulator Sensor* (*IMS*) Objekte ermöglichen der Anwendung Parameter vom Benutzer zu erfragen, und entsprechen somit am ehesten den klassischen 2D Bausteinen. Sie beinhalten keine einzelnen graphischen Repräsentationen sondern wählen selbständig, abhängig von der Laufzeitumgebung, eine geeignete Interaktionsform aus. In immersiven Umgebungen werden zur Interaktion mit den entsprechenden Elementen direkt *DMS* Objekte eingesetzt.

Die drei genannten Sensor-Ebenen bauen direkt aufeinander auf. Sie setzen die Technik der jeweils tieferen Stufe ein, um eine weitere Abstraktion zu schaffen.

4.1 Grundlagen

Die Auswahl der geeigneten Eingabe- und Ausgabegeräte ist ein wichtiger Schritt, um erfolgreich MR-Anwendungen zu entwickeln. Zum Beispiel kann es notwendig sein, die Kopfposition des Benutzers zu erfassen oder spezielle Hand-Gesten zu erkennen, um Applikationsabläufe steuern zu können. Die Auswahl der Eingabegeräte beeinflusst dabei entscheidend die Benutzbarkeit der Anwendung.

In den folgenden Abschnitten werden die gebräuchlichsten Interaktionsgeräte und Gerätegruppen vorgestellt und klassifiziert. Dabei werden nur die wichtigsten Klassen angesprochen. Eine ausführlichere Diskussion und Klassifizierung findet sich in [25][26].

In der Literatur findet sich eine Vielzahl von unterschiedlichen Kriterien zur Klassifikation von Eingabegeräten. Eines der wichtigsten ist die Anzahl der Freiheitsgrade (Degree of freedom/DOF). Dabei ist jeder Freiheitsgrad ein bestimmter, unabhängiger Weg, um einen absoluten oder relativen Sensor im Raum zu bewegen. Viele Geräte zum Messen und Verfolgen von Objekten (tracker) erfasst dabei drei Positionswerte und drei Rotationswerte. Diese geben zusammen sechs Freihatsgrade oder ein *six-DOF-Device*. Für die meisten Anwendungen ist der DOF ein Indikator für die Komplexität eines Geräts und bestimmt somit die Einsatzmöglichkeiten für unterschiedliche Interaktionstechniken. Ein weiteres Kriterium zur Klassifikation ist die Frequenz mit denen das Ein-/Ausgabegerät Daten liefert bzw. verarbeitet. Daten werden dabei entweder in diskreten Komponenten oder kontinuierlich geliefert. Diskrete Geräte liefern typischerweise Komponenten, die aus einem einzigen Datenwert bestehen (zum Beispiel einen Boolean-Wert oder ein Element aus einer Auswahl) auf der Grundlage einer Benutzerinteraktion. Diese werden oft zum Schalten eines Applikations-Modi oder zum Initiieren einer Aktion benutzt. Kontinuierliche Eingabegeräte generieren mehrere Datenpakete (zum Beispiel Bildschirmkoordinaten) als Reaktion auf eine Benutzerinteraktion.

Einige Geräte stellen Mischformen dar: sie ermöglichen Eingaben und versorgen den Benutzer gleichzeitig mit graphischer oder haptischer Rückkopplung oder sie verbinden mehrere Einzelgeräte in einer Einheit.

In den weiteren Abschnitten werden einige gebräuchliche Ein- und Ausgabegeräte auf der Gundlage dieser Klassifikation vorgestellt.

4.1.1 Interaktionsgeräte

Die Interaktionsgeräte ermöglichen dem Benutzer mit einem Computersystem und den bereitgestellten Anwendungen zu interagieren. Grundsätzlich kann man zwischen Eingabegeräten und Ausgabegeräten unterscheiden. Eingabegeräte ermöglichen dem Benutzer Kommandos und Daten zu erzeugen, und mit ihrer Hilfe ein Softwaresystem zu steuern. Ausgabegeräte stellen Informationen über den aktuellen Systemzustand dar.

Mit dem Aufkommen der interaktiven dreidimensionalen Computergraphik wurden eine Reihe neuartiger Interaktionsgeräte entwickelt. Ihnen ist gemeinsam, daß sie wesentlich mehr Freiheitsgrade besitzen und dadurch die direkte Interaktion im dreidimensionalen Raum ermöglichen. Besondere Bedeutung haben dabei Geräte, die Position und Orientierung im dreidimensionalen Raum messen, insgesamt also über sechs Freiheitsgrade verfügen. Diese Geräte werden in den folgenden Ausführungen als 6D-Geräte bezeichnet.

4.1.1.1 Eingabegeräte

Eingabegeräte sind physikalische Systeme, die Größen messen und an den Computer weiterleiten. Dabei lassen sich die Geräte in die folgenden Kategorien aufgeteilen:

- Geräte für Schreibtischumgebungen (desktop)
- Positions- und Orientierungsmesssysteme (tracker)
- 3D-Mäuse
- Medienströme

4.1.1.1.1 Geräte für Schreibtischumgebungen Alle Geräte dieser Klasse sind immobil. Das heißt, sie werden im allgemeinen auf der Schreibtischoberfläche eingesetzt und liefern entweder diskrete Werte (z.B. Knöpfe) oder *relative* Daten. Die kontinuierlichen Geräte dieser Klasse messen keine *absoluten* Positionen und Orientierungen, sondern nur Positionsänderungen.

Tastatur Die Tastatur ist ein Beispiel für ein tratitionelle Desktop-Eingabegerät, welche eine Menge an diskreten Komponenten bereitstellt (einen Menge an Knöpfen). Die Tastatur wird in vielen 3D-Desktop-Anwendungen eingesetzt, von Modellierungssystemen bis zu Computer-Spielen. Zum Beispiel

werden die Pfeiltasten oft zur Kameranavigation benutzt. Der Einsatz einer Standard-Tastatur ist jedoch in immersiven Umgebungen nicht praktikabel, da die Benutzer meistens stehen, wenn sie einen HMD der Stereo-Projektssysteme benutzen.

2D-Mäuse und Trackballs 2D-Mäuse sind ein weiteres klassisches Beispiel für Desktop-Eingabegeräte. Sie wurden vor allem in Systemen eingesetzt, die der *Windows, Icons, Menus und Pointers* (WIMP)[41] Metapher folgen. Ein Trackball ist mehr oder weniger eine umgedrehte Maus. Statt das ganze Gerät zu bewegen dreht der Beutzer nur die eingearbeitete Kugel. Ein Vorteil des Trackballs ist, dass er keinen festen Untergrund benötigt, was ihn für immersive Umgebungen interessant macht. Beide Geräte liefern eine relative 2D Position, die in 3D-Anwendungen zur Navigation und Interaktion genutzt werden kann.

Stift-basierte Eingabegeräte Stift-basierte Eingabegeräte gehören zu der gleichen Klasse von Eingabegeräten wie die 2D-Mäuse, benutzen aber einen anderen Formfactor. Weiterhin generieren sie keine relativen 2D Positionen sondern direkt 2D Pixelwerte, die von der Applikation ausgewertet werden können. Grosse Geräte sind für immersive Umgebungen ungeeignet, aber kleinere, wie zum Beispiel PDA-Geräte, können sehr effizient und elegant in immersiven Umgebungen eingesetzt werden.

Joysticks Joysticks stellen, analog zu den 2D-Mäusen und die Stift-basierten Eingabegeräten, gleichzeitig kontinuierliche 2D-Bewegungen und diskrete Knöpfe zur Verfügung. Es gibt jedoch einen entscheidenden Unterschied. Die meisten Anwendungen, die eine Navigation mit der Maus zur Verfügung stellen, stoppen, sobald die Maus nicht mehr bewegt wird. Anders beim Joystick: Konstante Auslenkung der Achsen werden in den meisten Fällen als konstante Bewegung in eine durch die Auslenkung bestimmte Richtung interpretiert. Für immersive Umgebungen sind Joysticks im Allgemeinen gut zu benutzen. Seit einigen Jahren gibt es auch Joysticks, die ihre Messdaten mithilfe von Funk übertragen und somit keine Kabelverbindung zu einem Rechner benötigen.

6-DOF-Geräte Die bisher vorgestellten Desktop-Geräte werden zwar für 3D-Anwendungen eingesetzt, aber wurden nicht speziell für diese Anwendungsgruppe entwickelt. Speziell für die Interaktion und Navigation in

Abbildung 4.1: Spacemouse Interaktionsgerät

3D-Anwendungen wurden spezielle Sechs-DOF Geräte enwickelt. Die gebräuchtlichsten sind die Spacemouse [80] (siehe Abbildung 4.1) und der Spaceball. Beide Geräte besitzen einen Griff, der mit der Hand verschoben und gedreht werden kann. Dieser ist bei der Spacemouse als flacher Puck gestaltet, während er beim Spaceball kugelförmig ist. Das vorwärts Drücken des Pucks einer Spacemouse leitet eine Vorwärtsbewegung ein. Der Benutzer kann durch das Verschieben und Verdrehen des Pucks oder der Kugel gleichzeitig Translationen und Rotationen steuern. Beide Geräte müssen beim Arbeiten fest auf dem Schreibtisch platziert werden, desshalb eigenen sie sich nicht für immersive Umgebungen.

4.1.1.1.2 Positions- und Orientierungsmesssysteme Für viele 3D-Anwendungen ist es wichtig, die Informationen über die Position einzelner pysikalischer Objekte oder des Benutzers zu messen. Das sind vorallem Geräte, die in Immersiven oder AR-Umgebungen zusammen mit Projektionssystemen [31][147] oder Datenbrillen (*Head Mounted Displays, HMD*) [40][9] zum Einsatz kommen, um direkt die Position und Orientierung zu messen. Hier handelt es sich um *absolute* Gerätedaten, die im Allgemeinen automatisch an die Anwendung weitergegeben werden, ohne dass der Benutzer den Messvorgang kontrolliert.

Bewegungsmesssysteme Zur Registrierung von Körperbewegungen werden Bewegungsmesssysteme, sogenannte *Motion Tracker* eingesetzt. Es gibt am Markt eine Vielzahl von Geräten und Gerätegruppen, welche mit unterschiedlichen Verfahren die Messungen vornehmen. Sie unterscheiden sich in der Reichweite, Latenz (Verzögerung zwischen dem Zeitpunkt, als die Bewegung aufgetreten ist und wann sie von der Anwendung registriert

Abbildung 4.2: Elektromagnetische Messsysteme eingesetzt in einer VR-Simulation

wird), Rauschen und Genauigkeit. Am gebräuchlichsten sind die folgenden Messverfahren:

Elektromagnetisch Elektromagnetische Systeme [101][7] bestehen aus einer elektrischen Magnetquelle, die ein Niederfrequenz-Feld aufbaut, und einer Reihe von Sensoren, die das erzeugte Magnetfeld messen. Die Messwerte werden über Kabel oder Funk an die Zentraleinheit des Systems übertragen. Diese berechnet Position und Orientierung jedes Sensors relativ zur Magnetquelle. Elektromagnetische Systeme werden schon lange in VR/AR-Applikationen eingesetzt (siehe Abbildung 4.2). Sie sind schnell und zuverlässig. Jedoch haben sie Probleme mit Störungen, die durch alle ferromagnetischen Objekte erzeugt werden. Sie waren bis vor kurzem nicht kabellos erhältlich.

Optisch Optische Trackingsysteme [117][116][1] ermitteln mithilfe von Videobildern die Position und Orientierung von Objekten. Dabei werden entweder markerlose oder markerbasierte Verfahren eingesetzt. Marker sind typischerweise kleine passive Kugeln oder Scheiben, die Licht in einem bestimmten Spektralbereich stark reflektieren oder aktive Lichtquellen, die Licht mit einer definierten Frequenz erzeugen. Ein System aus ein oder mehreren Kameras und einer Zentraleinheit erkennt zunächst die Markerposition im 2D-Bild und errechnet dann aus der Kombination von Bildern verschiedener Kameras die angenommene

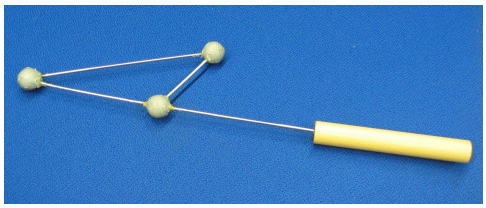

Abbildung 4.3: Kabelloses Interaktionsgerät mit optischen Markern

dreidimensionale Position des Markers im Raum. Um die Orientierung zu erkennen, müssen entweder drei oder mehr Marker fest miteinander verbunden werden (siehe Abbildung 4.3). Ein Optisches Trackingsystem ist immer kabellos und hat keine Probleme mit ferromagnetischen Störungen. Jedoch ist es anfällig für Verdeckungen. Wird ein Marker von zu vielen Kameras nicht mehr gesehen, kann die korrekte Position nicht berechnet werden.

Ultraschall Systeme, die Ultraschall zur Messung einsetzen, bestehen aus mindestens einer Audioquelle und einem Empfänger bzw. Mikrofon. Die Systeme werden zur Messung von Position und Orientierung eingesetzt [61] und sind vergleichbar mit elektromagnetischen Systemen. Vorteilhaft ist, dass die Daten mit höherer Genauigkeit gemessen werden können, jedoch haben sie im allgemeinen eine kurze Reichweite und geringe Messraten.

Mechanisch Mechanische Messsysteme besitzen eine rigide Struktur mit einer festen Anzahl an verbundenen mechanischen Gelenken und sind mit elektromechanischen Wandlern, wie zum Beispiel Potentiometern oder Achsenkodierern ausgestattet. Ein Ende der mechanischen Kette ist fest positioniert und das andere Ende wird mit dem Objekt verbunden, dessen Position oder Orientierung zu messen ist. Wenn sich das zu messende Objekt bewegt, verschiebt sich die mechanische Kette und dies kann wiederum über die elektromechnischen Wandler abgegriffen werden. Mechanische Messsysteme haben den Vorteil, dass sie im allgemeinen sehr genau sind und kaum Verzögerungen aufweissen. Auf der anderen Seite sind sie oft unförmig und behindern den Benutzer in seiner Bewegungsfreiheit.

Trägheit Trägheitsmesssysteme benutzten einen Vielzahl an Trägheitssensoren wie zum Beispiel Gyroskope zur Messung der

Abbildung 4.4: Datenhandschuh zur Messung von Fingerstellungen

Winkelgeschwindigkeit oder lineare Beschleunigunsmesser. Diese Geräte liefern abgeleitete Grössen (zum Beispiel Geschwindigkeiten oder Beschleunigungen) und somit ist es notwendig die einzelnen Messwerte zu integrieren, um absolute Positionen und Rotationen zu bekommen, was schnell zu Abweichungen und Messungenauigkeiten führen kann. Der Vorteil ist, das diese Geräte im Allgemeinen mit sehr hohen Abtastraten arbeiten.

Datenhandschuh In machen Anwendungen ist es wichtig, dass man detailierte Messdaten über die Hand des Benutzers bekommt. Dabei kann es notwendig sein, die Stellungen der Finger zueinander zu messen oder zu überwachen, ob die Fingerspitzen sich berühren. Der Datenhandschuh (*dataglove* [139]) (siehe Abbildung 4.4) ist ein VR-Standardgerät, das die Informationen liefern kann. Dabei messen bis zu 22 Sensoren die Stellung der Finger in Echtzeit. Aus diesen Messwerten kann ein dreidimensionales Modell der Hand errechnet werden. Durch Auswertung der Fingerstellung können Gesten erkannt und Handhabungsoperationen wie zum Beispiel Greifen und Zeigen nachgebildet werden. Sie sind sehr gut für immersive Umgebungen geeignet, da sie problemlos im Stehen genutzt werden können.

4.1.1.1.3 3D-Mäuse In dem vorhergenden Abschnitt wurden unterschiedliche Geräte zum Messen von Position und Orientierung vorgestellt. In vielen Anwendungsfällen werden diese mit anderen physikalischen Geräten wie zum Beispiel Knöpfen kombiniert, um flexiblere Eingabegeräte zu erhalten. Diese Kombinationen werden im allgemeinen als 3D-Mäuse bezeichnet. Sie sind dadurch gekenn-

zeichnet, dass sie ein 3D/6D-Bewegunsmesssystem mit einer Menge an diskreten Eingabekomponenten verbinden. Sie unterscheiden sich von klassischen 2D-Mäusen dadurch, dass der Benutzer sie durch den realen, physischen Raum frei bewegen kann und nicht an eine Tischoberfläche gebunden ist. Dadurch sind sie sehr gut in immersiven Umgebungen einsetzbar.

4.1.1.1.4 Medienströme Diese Geräte wurden nicht speziell für VR- oder AR-Systeme entwickelt. Sie liefern unabhängig von der Applikationsumgebung direkte Datenströme. Diese Ströme sind entweder Eingangsdaten für weitere Geräteklassen (zum Beispiel optische Tracker), oder sie werden direkt mit Elementen im Szenen-Graph des MR-Systems verbunden.

Videokameras Die Bilddaten von Videokameras werden nicht nur zur Berechnung von Position und Orientierung benutzt, sondern sie müssen dem MR-System auch direkt übermittelt werden. Besonders für AR-Anwendungen ist es notwendig, dass die Videoströme effizient eingelesen und verarbeitet werden können, damit sie synchron in den Bildgenerierungsprozess einfließen.

Mikrofone Akustische Ströme werden typischerweise nur in verteilten oder Mehr-Benutzer MR-Anwendungen direkt eingesetzt, um Elemente im Szenengraph zu versorgen. Viel häufiger dienen sie als Quelle für Sonar-basierte Trackingsysteme und Spracherkennungsverfahren. Mit dem Aufkommen von leistungsfähigen Spracherkennungssystemen ist eine neue und mächtige Mensch-Maschine Schnittstelle entstanden, die auch in der Virtuellen Realität angewendet wird. Gesprochene Befehle erlauben die direkte Manipulation der virtuellen Umgebung.

4.1.1.2 Ausgabegeräte

Obwohl Bildschirme und Projektionssysteme die wichtigsten Ausgabegeräte für MR-Applikationen sind, sollen sie an dieser Stelle nicht betrachtet werden. Die Bildsynthese und somit die graphische Ausgabe ist im Allgemeinen direkt in das MR-System integriert und muss somit gesondert behandelt werden.

An dieser Stelle werden kurz die Ausgabegeräte vorgestellt die analog zu den zuvor vorgestellten Eingabegeräten als externe Komponenten betrachtet werden können.

4.1.1.2.1 Haptische Ausgabegeräte Nach [148] erfüllen haptische Ausgabegeräte (auch haptische Displays genannt) zwei Aufgaben:

1. Darstellung bzw. Vermittlung von haptischen Reizen und Positionsänderungen.

2. Messung von Position und Kontaktkräften der Hand oder anderer Körperteile, sowie deren zeitlicher Veränderung.

Im Unterschied zu den bisher in dieser Arbeit betrachteten Interaktionsgeräten kommunizieren haptische Ausgabegeräte bidirektional mit der Anwendung. Die vom haptischen Display ermittelte Position ist an die Anwendung zu übertragen, da diese von der zu simulierenden Kontaktkraft von der Position der Hand bzw. eines Instrumentes abhängig ist. Das gleiche gilt für die vom Gerät gemessene Kontaktkraft. Für die Anwendung wiederum kann es erforderlich sein, die Hand oder das Instrument an eine bestimmte Position zu bringen, um die vom Gerät darzustellende Kraft zu steuern. Dies erfordert eine Datenübertragung von der Anwendung an das haptische Ausgabegerät.

Bei der Kommunikation zwischen Anwendung und haptischen Displays sind besondere Anforderungen an die Geschwindigkeit der Datenübertragung zu erfüllen. Während bei Eingabegeräten eine Übertragungsrate entsprechend der Rate der Bildgenerierung von 50 Hz ausreicht, ist für haptische Displays eine Frequenz von bis zu 10.000 Hz erforderlich [148].

Ein weiteres Problem entsteht bei der Anwendung selbst: Diese muss in der Lage sein, die physikalische Simulation mit derselben Frequenz durchzuführen.

4.1.1.2.2 Kraftrückkopplung Kraftrückkopplungsgeräte vermitteln die erzeugte Kontakt- oder Reaktionskraft, indem Körperteile (Finger, Hand, Arm) an eine andere Position gebracht werden [148]. Der Benutzer muss eine Gegenkraft aufwenden, um die gewünschte Position aufrecht zu halten. Meist werden kleine leistungsfähige Elektromotoren verwendet. Besitzen diese Motoren eine sehr hohe Dynamik, können auch Vibrationen erzeugt werden. Beispielhaft soll hier das PHANToM (Personal HAptic iNTerface Mechanism) [85] erwähnt werden: Der Anwender nimmt die Kräfte entweder an der Fingerspitze die er in einen Fingerhut steckt wahr, oder durch einen Stift, der in den Fingerhut gesteckt wird. Das Gerät besitzt drei Freiheitsgrade, die über drei Elektromotoren realisiert werden. Die Maximalkraft ist mit 10 N, die kontinuierlich Rückkopplungskraft mit 1.5 N angegeben. Die verbleibende Reibungskraft im Leerlauf ist 0.1 N [148].

4.1.1.2.3 Taktile Rückkopplung Wahrnehmung wie das Ertasten von Oberflächentexturen kann mit sogenannten taktilen Displays erzeugt werden. Taktile Displays reizen die Rezeptoren, die für den Tastsinn verantwortlich sind, über direkten Hautkontakt.

4.1.2 System zur Geräteabstraktion

Die Gerätehersteller liefern im allgemeinen eine gerätespezifische Bibliothek die von Anwendungsentwicklern direkt benutzt werden kann. Ist während der Entwicklung ein Gerät zu wechseln, zum Beispiel um statt eines elektromagnetischen einen optischen Tracker einzusetzen, sind aufwendige Anpassungen notwendig, da die meisten Geräte unterschiedliche Schnittstellen bereitstellen.

Um diesen Aufwand zu minimieren wurden einige Systeme [34][112][75][107][50][120][39][114] entwickelt und vorgestellt, die Abstraktionen für Geräte bereitstellen.

Diese geräteunabhängigen Schnittstellen erlauben eine Rekonfiguration der Applikation und vereinfachen den Wechsel von Geräten wesentlich. Die Systeme unterscheiden sich dabei in den bereitgestellten Geräteklassen und Spezifikationen, verfolgen aber ein gemeinsames Ziel: Die Integration von Ein- und Ausgabegeräten unabhängig von spezifischen Schnittstellen möglichst flexibel und einfach zu gestalten.

Darüber hinaus bieten diese Systeme Merkmale, welche über die eigentliche Abstraktion von Geräten hinausgehen:

Datenfilter Einige Systeme [34][107] bieten Datenfilter neben den Geräteabstraktionen an. Diese Datenfilter können zwischen den virtuellen Geräten und der Anwendung eingesetzt werden um zum Beispiel Messungen zu glätten, zu entstören oder bei fehlenden Daten vorherzusagen.

Netzwerktransparenz Um Ein- und Ausgabegeräte nutzen zu können, die physikalisch an der Applikationsmaschine angebracht sind, erlauben moderne Systeme [34][112][107][50][75] netzwerktransparenten Zugriff auf Geräte. Das bedeutet, dass die Anwendung transparent auf lokale und über ein Netzwerk erreichbare Geräte zugreifen kann.

Automatisches Auffinden von Netzwerkgeräten Netzwerktransparente Systeme stellen fast immer eine Client/Server Architektur zur Verfügung. Die MR-Applikation entspricht dabei der Client-Anwendung, welche die Daten

von einem Geräte-Server liest. Im Allgemeinen muss der Client (in diesem Fall die VR-Anwendung) den Server kennen, um darauf zugreifen zu können. Um die Konfiguration in Netzwerken zu vereinfachen bieten einige Systeme [75][34] abstrakte Namensräume für Geräte oder Datenströme an, die unabhängig von dem konkreten Server-System sind.

Rekonfigurationen zur Laufzeit In einigen Systemen [34] ist die Konfiguration der virtuellen Geräte und Filter nicht statisch, sondern kann zur Laufzeit graphisch-interaktiv verändert werden.

In dieser Arbeit wurde keine weitere Abstraktion entwickelt. Es wurden bestehende Systeme so integriert, dass die angebotenen virtuellen Geräte bzw. Ströme als Quellen und Senken benutzt werden können.

4.2 Sensoren für Datenströme

Die meisten Systeme, die kein festes Interaktionsmodell mit einem Eingabegerät verbinden, sondern deren Daten möglichst direkt bereitstellen wollen, deklarieren zu diesem Zweck abstrakte Geräteklassen die unterschiedliche Gruppen abbilden (siehe 4.1.2). Aufbauend auf diesen abstrakten Klassen werden konkrete Implementierungen für unterschiedliche physikalische Geräte angeboten.

Dieser Ansatz erlaubt zwar die Daten direkt in die Anwendung einzubringen, hat aber den Nachteil dass für neuartige Geräte, die nicht in die definierten Gruppen passen, neue Basisstrukturen geschaffen werden müssen. Darüber hinaus sind Verschmelzungen von Sensordaten nicht ausreichend abbildbar.

Um nicht nur von physikalischen Geräten sondern auch von deren Klassifikation unabhängig zu sein wird an dieser Stelle ein anderer Ansatz verfolgt:

- Die Sensoren sind möglichst generisch und sind für einen weites Anwendungsfeld einsetzbar. Dazu müssen sie das Einlesen und die Ausgabe von Daten unterstützen.
- Die Abstraktion beschränkt sich nicht auf den Datenaustausch mit physikalische Geräten, sondern ist offen für Datenfilter und der allgemeinen Kommunikation mit geräteähnlichen Applikationen.
- Die Anwendungen sind immer plattformunabhängig. Um dies zu erreichen, sind die Sensoren unabhängig von den unterschiedlichen Plattformen

und Geräteklassifikationen. Die Sensoren definieren was die Applikation benötigt, jedoch nicht welches Gerät oder Geräteklasse diese Information bereitstellt.

Die hier vorgestellten Sensorknoten abstrahieren keine einzelnen Geräte oder Geräteklassen sondern typisierte Datenströme. Ein Datenstrom ist ein Strom von Werten der die Grenzen des Systems durchbricht und entweder von Innen nach Außen oder von Außen nach Innen gerichtet ist. Jeder einzelne Strom transportiert nur Werte oder Gruppen von Werten eines Datentyps. Als Schnittstellen für diese Ströme werden vom System Sensorknoten bereitgestellt.

Die Ein- und Ausgabegeräte und ihre Schnittstellen, welche von den externen Systemen (siehe 4.1.2) bereitgestellt werden, sind nicht direkt auf Sensor-Knoten abgebildet. Jeder einzelne Parameter ist auf jeweils einem Knoten abgebildet. Ein Joystick-Interface mit vier Knöpfen und zwei Achsen stellt somit zum Beispiel vier SFBool und zwei SFFloat Ströme bereit, die bei Bedarf mit entsprechenden Knoten verbunden werden.

Diese Sensoren für Datenströme (*Data Stream Sensor/DSS*) sind wie folgend definiert:

```
xSensor {
  SFString  []              label       ''''
  SFBool    [in,out]        repeat      FALSE
  SFBool    [in,out]        transform   FALSE
  SFBool    [in,out]        cacheSize   0
  x         [in,out]        value
}
```

Das konkrete Sensor-Interface für binäre Ströme in dem SFBoolSensor Interface ist zum Beispiel wie folgt definiert:

```
SFBoolSensor {
  SFString   []             label       ''''
  SFBool     [in,out]       repeat      FALSE
  SFBool     [in,out]       transform   FALSE
  SFBool     [in,out]       cacheSize   0
  SFBool     [in,out]       value
}
```

Für jeden elementaren Feldtypen (siehe Abschnitt 3.3.1) stellt das System genau einen Sensortyp (siehe Tabelle 4.1) bereit.

Die Feldwerte als Parameter spezifizieren das Laufzeiterhalten der Sensoren:

label Definiert einen eindeutigen Namen für den Strom im Systemraum. Dieser Name wird benutzt, um den Strom mit externen Quellen und Senken zu verknüpfen.

repeat Definiert, das der Sensor in jeden Zyklus Daten liefern soll, auch wenn keine neuen Daten anliegen. Liefert zum Beispiel ein externes Gerät keine neuen Werte wird der letzte Wert erneut verschickt.

transform Benutzt die globale Transformation des Szenengraphen um den aktuellen Wert zu transformieren. Dies ist nur dann sinnvoll, wenn die Werte eine Koordinate im Raum definieren. 1D (SFInt32,MFInt32,SFFloat,MFFloat), 2D (Vec2f,Vec2d), 3D (Vec3f,Vec4f,SFColor) und 4D (Vec4f,Vec4d,SFColorRGBA) werden direkt mit der aktuellen akkumulierten Transformationsmatrix transformiert. Matrizen (SFMatrix,MFMatrix) werden als lokale Transformation interpretiert und mit der globalen Matrix akkumuliert.

cacheSize Definiert die Anzahl der Elemente die gepuffert werden. Der Wert von -1 definiert, dass alle Werte gepuffert werden. Mit einem Wert von 1 wird nur der letzte Wert verarbeitet.

Diese Sensoren definieren somit nur typisierte Datenströme. Sie sind an keine Interaktions- oder Navigationsmethoden gebunden und bilden die unterste Schicht für das hier vorgestellte Sensorkonzept.

Diese Sensoren sind vor allem dann sinnvoll, wenn die Eingangsdaten in der Anwendung direkt und ohne weitere Manipulation durch den Benutzer verarbeitet werden sollen.

Dazu zwei Beispiele:

Videobilder in AR-Anwendungen In vielen typischen AR-Anwendungen werden Videodaten von einer angeschlossenen Kamera gelesen, möglicherweise für die Bestimmung der Position und Orientierung direkt verarbeitet und dann als Hintergrund in der eigentlich Anwendung zur Darstellung gebracht (siehe Abbildung 4.5). Eine entsprechende Kodierung ist wie folgt anzuwenden:

Einzelen Daten	Datenverbund	Datentypen
SFStringSensor	MFStringSensor	Text
SFBoolSensor	MFBoolSensor	Binäre Daten
SFFloatSensor	MFFloatSensor	Skalare Werte mit einfacher Genauigkeit
SFDoubleSensor	MFDoubleSensor	Skalare Werte mit doppelter Genauigkeit
SFInt32Sensor	MFInt32Sensor	Integer Werte
SFTimeSensor	MFTimeSensor	Zeitwerte
SFColorSensor	MFColorSensor	RGB kodierte Farbwerte
SFColorRGBASensor	MFColorRGBASensor	RGBA kodierte Farbwerte
SFVec2fSensor	MFVec2fSensor	Vektor mit zwei Komponenten und einfacher Genauigkeit
SFVec3fSensor	MFVec3fSensor	Vektor mit drei Komponenten und einfacher Genauigkeit
SFVec4fSensor	MFVec4fSensor	Vektor mit vier Komponenten und einfacher Genauigkeit
SFVec2dSensor	MFVec2dSensor	Vektor mit zwei Komponenten und doppelter Genauigkeit
SFVec3dSensor	MFVec3dSensor	Vektor mit drei Komponenten und doppelter Genauigkeit
SFVec4dSensor	MFVec4dSensor	Vektor mit zwei Komponenten und doppelter Genauigkeit
SFRotationSensor	MFRotationSensor	Rotationsdaten als Quaterion kodiert
SFPlaneSensor	MFPlaneSensor	Ebenen
SFMatrix4f	MFMatrix4f	Matrizen mit 4x4 Komponenten und einfacher Genauigkeit
SFMatrix4d	MFMatrix4d	Matrizen mit 4x4 Komponenten und doppelter Genauigkeit
SFImage	MFImage	1D/2D und 3D RGBA Bilder
SFSound	MFSound	PCM kodierte Audiodaten

Tabelle 4.1: Sensortypen für direkte Datenströme

Abbildung 4.5: SFImageSensor Anwendung in einer AR-Umgebung

Abbildung 4.6: Fahrsimulator als Anwendung von DSS Objekten

```
ImageBackground {
  ...
  image DEF image PixelTexture { ... }
  ...
}
DEF frame SFImageSensor {
  label "frame"
}
ROUTE frame.value_changed TO image.set_image
```

Steuerdaten in Fahrsimulation In dieser Anwendung (siehe Abbildung 4.6) werden die Messwerte des Lenkrades und die Pedalstellungen für Gas und Bremse gemessen und zur Laufzeit auf drei SFFloat Sensoren übertragen. Die Ausgänge

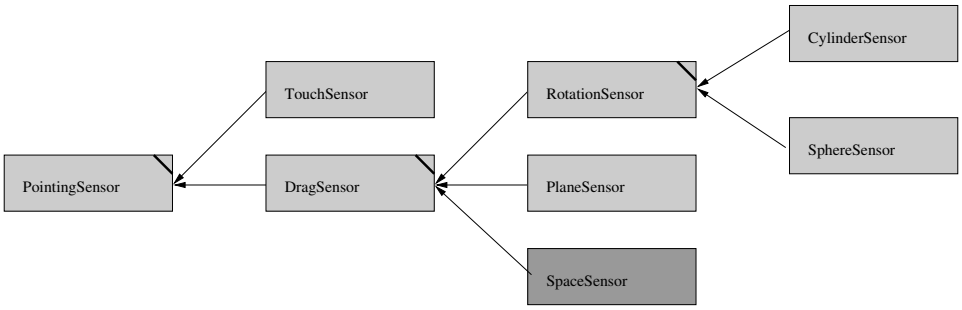

Abbildung 4.7: Vererbungshierachie der X3D-PointingSensor-Knoten

der Sensoren sind direkt mit der Simulationseinheit verbunden.

4.3 Sensoren zur direkten Manipulation

Sensoren zur direkten Manipulation (*Direct Manipulation Sensor, DMS*) greifen die Ideen der *PointingSensor*-Knoten der X3D Spezifikation [143] auf und erweitern diese für immersive Umgebungen. Dabei werden Grundlagen aus [12][125][124] übernommen.

Die Gruppe der X3D Pointingsensoren (siehe Abbildung 4.7) definieren über ihre Vaterbeziehung einen Teilgraphen der Szene, dessen Selektion (*TouchSensor)* oder Transformation (*DragSensor*) registriert wird.

Die *PointSensor, DragSensor und RotationSensor* Typen sind abstrakt und können somit nicht direkt instanziert werden. Der *TouchSensor* registriert Berührungen des Benutzers. Der *PlaneSensor* registriert die Verschiebung eines Objektes in der Ebene. Die *CylinderSensor* und *SphereSensor* reagieren auf Rotation die einer projizierten Verschiebung auf einen Zylinder bzw. auf einer Kugel entsprechen.

4.3.1 Sensoren zur Selektion eines Teilgraphen

Der *TouchSensor* ist sicherlich der einfachste Sensor in der X3D Spezifikation. Die X3D Semantik beschränkt sich auf 2D-Maus Interaktion. Sie verlangt, dass der Sensor registriert, wenn der Maus-Zeiger “über” dem Teilgraphen ist. Eine Selektion kann daraufhin mit der linken Taste der Maus erfolgen.

In immersiven Umgebungen sind die Begriffe “über” dem Objekt und “selektiert” nicht direkt umsetzbar. Aus diesem Grund wird an dieser Stelle ein neuer

Knotentyp eingeführt:

Der *UserBody* ist eine Spezialisierung des *Group*-Knoten und kann selbst einen Teilgraphen definieren. Dieser Teilgraph wird als virtuelles Körperteil des Benutzers interpretiert und ist die Grundlage für alle weiteren strahl- und kollisionsbasierten Interaktionsmodelle. Der Knoten besitzt weiterhin ein SFBool Feld *hot*, welches den *selection*-Zustand markiert. Die Schnittstelle des Knoten lässt sich folgendermaßen definieren:

```
BodyPart : Group {
   ...
   MFNode  children []
   SFBool  hot      FALSE
   ...
}
```

Der *BodyPart*-Knoten beinhaltet keine lokale Transformation. Er kann aber wie jeder andere *Group*-Knoten als Kind eines *Transform*-Knotens verwandt werden. Dieser *Transform*-Knoten kann wiederum mit *DDS*-Sensoren verbunden werden, um neue Transormationswerte zu erhalten. Über die Anbindung eines entsprechenden *DDS*-Sensors ist ebenfalls der hot Zustand der Gruppe zu steuern. Ein Knopf auf einem FlyStick [1] kann direkt über einen *SFBoolSensor DDS* Knoten mit dem hot-Field verbunden sein.

Der BodyPart Typ ist kein *Bindable* sondern kann beliebig oft in der Szene genutzt werden, um Teilgraphen als Köperteile zu deklarieren. Dies wird vor allem für mehrhändige Interaktionsmodelle benötigt.

In den nachfolgenden Teilabschnitten wird die Funktionsweise des Touchsensors in 2D und Immersiven Umgebungen erläutert, wobei nicht auf alle Einzelheiten der X3D Spezifikation [143] eingegangen wird.

4.3.1.1 2D-Interaktion

Wenn der 2D-Mauszeiger über eine Geometrie positioniert wird, welche mit einem *TouchSensor* assoziert ist, generiert der Sensor-Knoten eine $isOver(true)$ Nachricht. Wenn der Mauszeiger sich nicht mehr über der Geometrie befindet, wird eine $isOver(false)$ Nachricht erzeugt. Wird die linke Maustaste gedrückt, solange sich der Zeiger über dem Objekt befindet, wird eine $isActive(True)$ Nachricht versandt. Wird die Maustaste losgelassen wird ein $isActive(false)$ generiert. Um

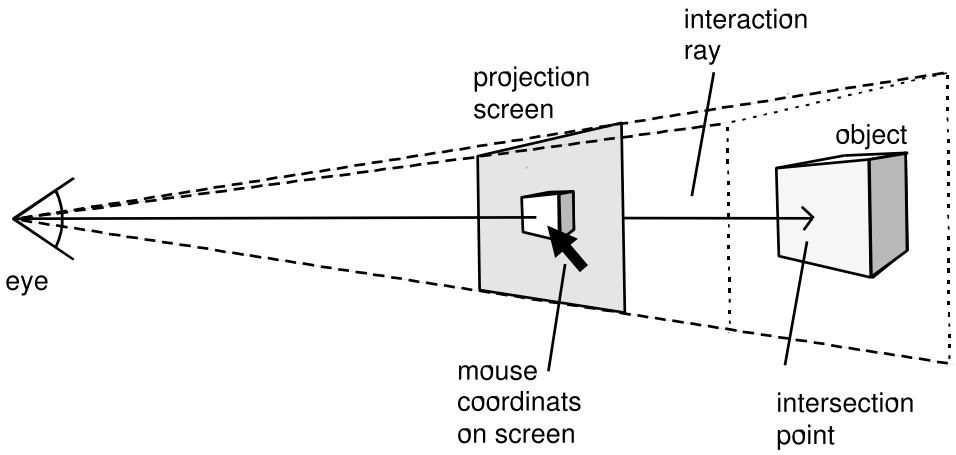

Abbildung 4.8: 2D-Mouse Selektion

zu prüfen, ob sich der Zeiger über der Geometrie befindet, wird abhängig von den aktuellen Kameraparametern und der Mausposition ein Strahl berechnet und durch Traversierung des Szenegraphen mit den Objekten geschnitten (siehe Abbildung 4.8).

4.3.1.2 Immersive Interaktion

In der immersiven Umgebung hat der Benutzer direkt die Möglichkeit *TouchSensor*-Teilbäume zu berühren. Dies wird mithilfe einer Kollisionserkennung ermittelt. Dazu werden alle *BodyPart*-Teilbäume gegen alle *Sensor*-Teilbaume getestet.

Das Problem ist jetzt, dass es nicht möglich ist, mit Sensoren auf Distanz zu interagieren. Um zu verhindern, dass der Benutzer immer zuerst zu den Sensoren navigieren muss um sie auszulösen, wird an dieser Stelle ein zweistufiges Verfahren vorgeschlagen. Befindet sich der Benutzer weiter als die virtuelle Armlänge des Avatars von dem Sensor entfernt, wird ein Strahl zur Selektion eingesetzt. Unterschreitet der Benutzer diese Grenze, wird automatisch auf Kollision geprüft.

4.3.1.2.1 Selektion auf Distanz Die Strahlgeometrie wird durch Position und Orientierung des *BodyPart*-Objektes bestimmt (siehe Abbildung 4.9). Wenn der Strahl einen Teil der Geometrie trifft, welcher mit einem Sensor assoziiert ist, dann versendet der *TouchSensor* ein $isOver(true)$. Wenn der Strahl sich nicht mehr länger mit der Geometrie schneidet, dann generiert der Knoten eine $isOver(false)$ Nachricht (siehe Abbildung 4.10). Schneidet der Strahl eine assoziierte Geometrie

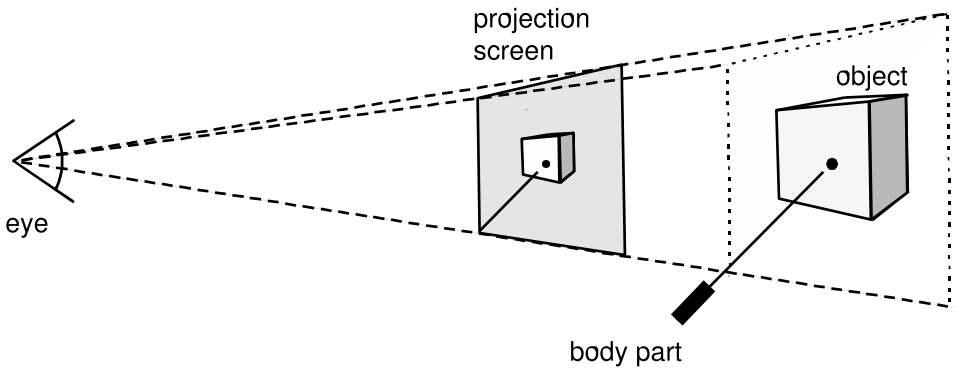

Abbildung 4.9: Strahlbestimmung zur immersiven Interaktion mit einem PointingSensor

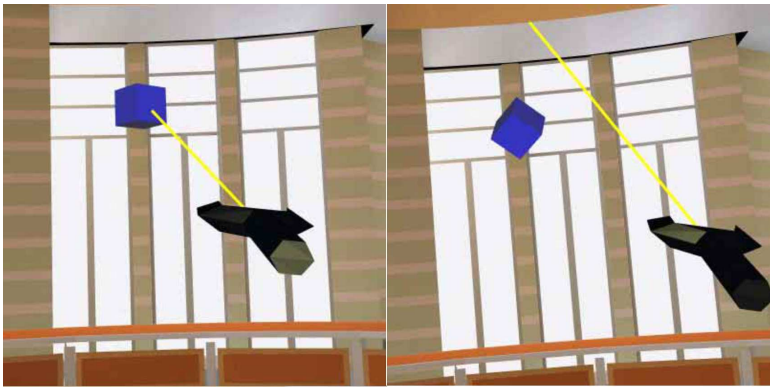

Abbildung 4.10: Interaktion mit einem TouchSensor auf Distanz

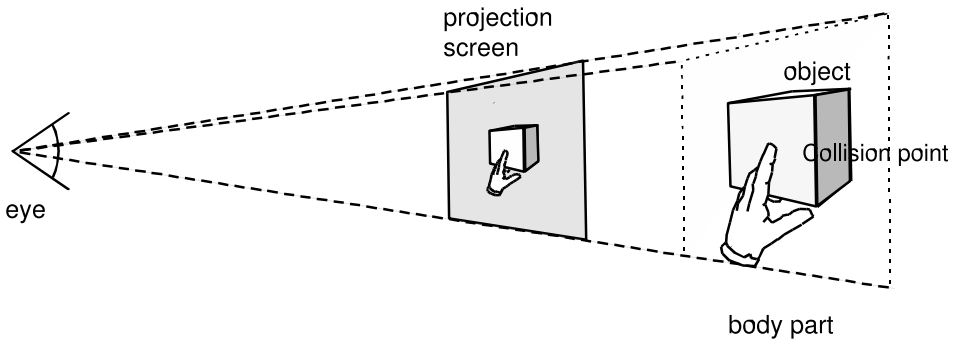

Abbildung 4.11: Kollision und Aktivierung des Sensors im Nahbereich

und das hot-Field des entsprechenden *BodyParts* wird auf den Wert $true$ gesetzt, dann versendet der Sensor, analog zum Klicken mit der linken Taste im 2D-Modell, ein $isActive(true)$. Wenn das hot-Feld auf the Wert $false$ zurückspringt, dann versendet der Sensor ein $isActive(false)$.

4.3.1.2.2 Selektion im Nahbereich Für die Interaktion im Nahbereich werden alle Sensor und *BodyPart*-Teilbäume bei der Kollisionserkennung angemeldet und paarweise gegeneinander getest. Kollidieren die Teilbäume, so wird ein $isOver(true)$ versandt. Durchdringen sich die Geometien der Teilbaume nicht mehr, so schickt der Sensor ein $isOver(false)$ Nachricht (siehe Abildung). Die Aktivierung des Sensors und somit die Generierung der $isActive(true)$ bzw. $isActive(false)$ Nachricht wird anlog zu der Selektion auf Distanz mithilfe des BodyPart Feldes hot gesteuert.

4.3.2 Sensoren zur Transformation eines Teilgraphen

Für die Transformation werden vier konkrete Sensor-Typen benutzt, die Spezialisierungen *DragSensor*-Typen sind. Die Schnittstellen und Grundlagen der *SphereSensor, PlaneSenor* und *CylinderSensor* werden aus der X3D Spezifikation [143] übernommen. Das elmentare Verhalten der X3D Sensoren ist in der Spezifikation festgeschrieben. Sie definiert wie die Bewegungen des Zeigegerätes über dem Objekt, abhängig von seiner virtuellen Gestalt, ausgewertet wird.

Die X3D Sensoren beschränken die Interaktion immer auf die virtuelle Gestalt des Sensors und sind somit für die freie 3D oder 6D Interaktion nicht geeignet.

Aus diesem Grund wird hier der *SpaceSensor* eingeführt. Dieser Sensor erlaubt es Objekte frei durch den Raum zu ziehen, und generiert ensprechend der aktuellen Transformation $translationChanged(vec3f)$ und $rotationChanged(rot)$ Nachrichten.

Die virtuelle Gestalt des Sensors ist definiert durch den Sensortypen und dem initialen Punkt auf der Geometrie an welchen der Sensor aktiv wurde. Es werden so lange Nachrichten (*Events*) generiert wie das Zeigegerät über den Sensor gezogen (*draged*) wird.

Die X3D Spezifikation diskutiert nur eine 2D-mausbasierte Implementierung. Diese wird im nächsten Abschnitt kurz vorgestellt. Im Folgenden werden die Ideen aufgegriffen und für immersive Umgebungen interpretiert und erweitert.

4.3.2.1 2D-Interaktion

Die drei *DragSensor*-Knoten verändern analog zu dem *Touchsensor* seinen $over$ und $active$ Zustand und versenden ensprechende $isOver(bool)$ und $isActive(bool)$ Nachrichten (siehe Abschnitt 4.3.1.1).

Alle *DragSensor*-Knoten $trackPointChanged(vec3f)$ erzeugen Nachrichten solange der Sensor aktiv ist. Bei der Ziehbewegung mit der Maus werden vom aktiven Sensor Koordinaten auf der virtuellen Sensoroberfläche berechnet und versendet.

4.3.2.2 Immersive Interaktion

Für die immersive Interaktion wird der *BodyPart*-Knoten (siehe Abschnitt 4.3.1) eingesetzt, um Transformation und Gestalt der virtuellen Hand festzulegen. Analog zu der Interaktion mit *TouchSensor*-Knoten werden, abhängig von der Distanz zwischen *Avatar* und *Sensor*-Geometrie, zwei unterschiedliche Methoden eingesetzt.

4.3.2.2.1 Transformation auf Distanz Auf Distanz werden Strahlen zur Interaktion eingesetzt, deren Geometrie durch Position und Orientierung des *BodyPart*-Objektes bestimmt (siehe Abbildung 4.9).

Bei *PlaneSensor* werden die Strahlen direkt auf die Bildebene projiziert und mit $trackPointChanged(vec3f)$ Nachrichten zur Verfügung gestellt. Für den *SpaceSensor* werden für alle Transformationen des BodyPart-Objektes über aktive Sensoren unmittelbar $translationChanged(vec3f)$ und

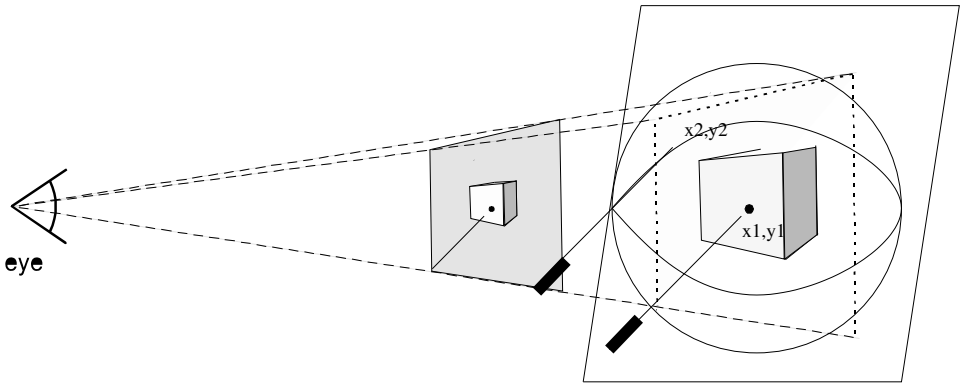

Abbildung 4.12: Projektionsfläche zur Bestimmung der Sensorrotation

$rotationChanged(vec3f)$ Werte versandt.

Für die Berechnung des Versatzes für *SphereSensor* und *CylinderSensor* wird die Differenz zwischen der initialen Position und der aktuellen Position auf einer Ebene benutzt.

Die Projektionsebene definiert eine Ebene, die das Zentrum der Sensorgeometrie schneidet und deren Normale in Richtung des Betrachters weist. Der initiale Schnittpunkt mit der Geometrie bestimmt den Radius für die weitere Rotation. Der Benutzer bewegt mit der Hilfe eines *DDS* Sensors den *BodyPart* um den neuen Projektionspunkt zu bestimmen (siehe Abbildung 4.12).

Die initiale Wert für die Orientierung wird mit dem initialen Schnittpunkt assoziiert. Die Differenz zwischen dem aktuell projizierten Wert (x_c, y_c) und dem initialen Wert (x_i, y_i) wird benutzt, um die neue Rotationsanteile zu bestimmen:

$$\alpha = \frac{\pi}{2}\frac{(x_i - x_c)}{\rho}$$

$$\phi = \frac{\pi}{2}\frac{-(y_i - y_c)}{\rho}$$

Die neuen Orientierungswerte (α, ϕ) werden zuerst bestimmt. Anschließend werden diese mit den initialen Werten summiert und daraufhin versendet.

4.3.2.2.2 Transformation im Nahbereich Für die Interaktion im Nahbereich wird die Geometrie des Sensors direkt mit dem der *BodyPart* Geometrie kollidiert (siehe Abschnitt 4.3.1.2.2). Für die Interaktion mit dem *PointSensor* wird die ak-

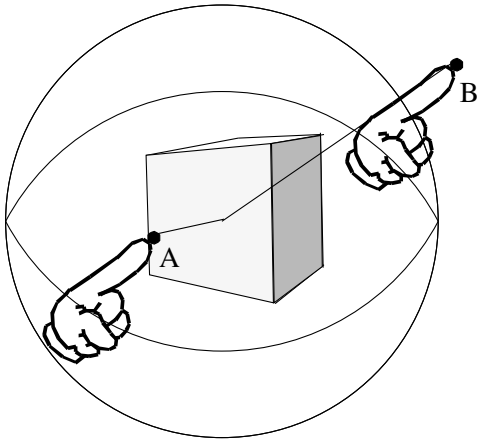

Abbildung 4.13: SphereSensor Interaktion im Nahbereit

tuelle 3D-Position auf die Sensorebene projiziert. Der *SpaceSensor* verarbeitet 3D Koordinaten. Somit kann die *Bodypart* Position direkt in das lokale Koordinatensystem transformiert und versendet werden.

Für die *SpaceSensor* und *CylinderSensor*-Knoten wird keine Hilfsebene generiert, sondern die neuen Rotationsanteile direkt aus der aktuellen Position des *BodyParts* berechnet. Dazu wird ein normalisierter Vektor bestimmt, der vom Zentrum der Sensorgeometrie zu der aktuellen *Bodypart* Position zeigt. Dieser Vektor wird als Quaternion, dessen Rotationsanteil Null ist, interpretiert und zur Berechnung der Rotationsanteile benutzt.

In Abbildung 4.13 wird der *BodyPart* von Position A zur Position B bewegt und zieht den aktiven Sensor direkt. Dabei ist es möglich den *BodyPart* frei in der Szene zu bewegen, um den Sensor um einen Punkt zu rotieren.

4.4 Sensoren zur indirekten Manipulation

Die bisher vorgestellten *DDS* und *DMS* Sensoren werden vorzugsweise für Interaktionsprozesse eingesetzt die nicht an einzelne Aufgaben gebunden sind. Die DDS Sensoren transportieren zustandslos und beständig Daten zwischen dem Systemraum und der angebunden Geräteabstraktion. Die DMS Sensoren registrieren Selektion und Transformation von Teilbäumen aber berücksichtigen dabei keinen expliziten *Start* bzw. *Ende* der Interaktion. Beide Sensoren besitzen ein Zustandsfeld *enabled* womit ihre Funktionalität unabhängig von dem aktuellen Zustand freigegeben werden kann. Dies ist nicht zu verwechseln mit dem kontrollierten

Durchlaufen von *Start/Eingabe/Ende* Zuständen.

Wenn es darum geht, dass die Anwendung für die weitere Abarbeitung konkrete Eingaben vom Benutzter benötigt, ist eine weitere Abstraktion notwendig. Aus diesem Grund werden an dieser Stelle die Sensoren zur indirekten Manipulation (*Indirect Manipulator Sensor*, IMS) eingeführt. Diese Sensoren werden an dieser Stelle als indirekt bezeichnet, da sie nicht explizit mit einer geometrischen Repräsentation verbunden sind dessen virtuelle Repräsentation der Benutzter direkt manipuliert. Die Sensoren bestimmen nicht wie die DMS-Sensoren einen Teil der Szene als Interaktiv und überlassen den Benutzter ob und wann er damit interagiert, sondern definiert vielmehr eine Anfrage der Anwendung an den Benutzer.

Die IMS-Sensor-Knoten bilden jeweils eine Basisinteraktionsaufgabe (*Basic Interaction Tasks*, BIT)[41] auf eine der folgenden Benutzerinteraktionen ab:

- Positionierung
- Selektion
- Quantifizierung
- Text
- Rotation

Grundsätzlich sind die Basisinteraktionsaufgaben unabhängig von der Interaktionsumgebung definiert. Dennoch sind einige Untersuchungen und Definitionen notwendig, um sie für VR-Umgebungen einsetzen zu können [102]. Christian Knöpfle hat in [69] zu diesem Zweck “Logische Interaktionsaufgaben” eingeführt. Die Interaktionsaufgaben können je nach Laufzeitumgebung auf unterschiedliche graphisch-interaktive Elemente abgebildet werden. Für Desktop VR-Umgebungen werden 2D-Window-Elemente eingesetzt, für immersive Umgebungen geometrische Repräsentationen, welche der Benutzter direkt mit den virtuellen Geräten manipulieren kann.

Die Unterteilung in 2D- und 3D-Interaktionselementen wird in dieser Arbeit beibehalten, jedoch wird die 3D-Repräsentation nicht direkt mit Interaktionsgeräten verbunden. Um die Flexibilität noch weiter zu erhöhen werden sie zur Laufzeit automatisch aus DMS Sensoren aufgebaut, die wiederum über 2D- oder 3D-Interaktionsmethoden (siehe Abschnitt 4.3) gesteuert werden (siehe Abbildung 4.14).

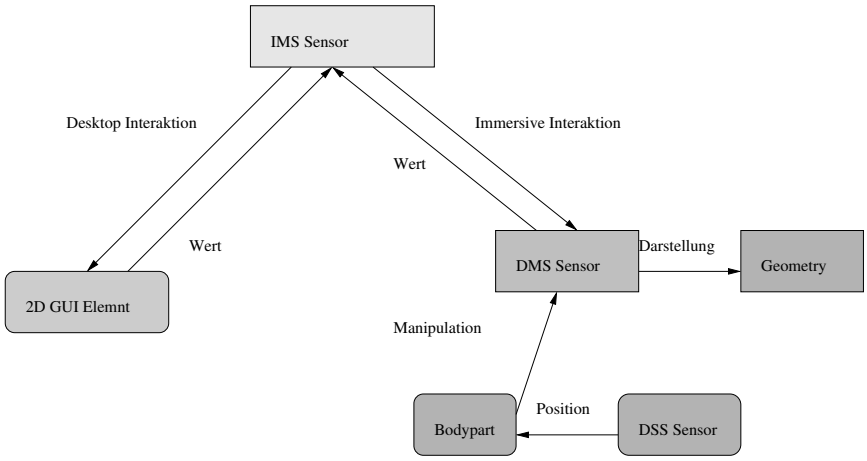

Abbildung 4.14: IMS Sensor Repräsentation für Desktop- und Immersive-Umgebungen

Alle IMS Sensoren sind als Spezialisierung der abstrakten *TaskSensor*-Knoten implementiert. Die Schnittstelle des Basisknoten ist für alle konkreten IMS Sensoren gültig:

```
TaskSensor : Sensor {
...
  SFString   [in,out]  name       ''''
  SFBool     [in,out]  viewSpace  TRUE
  SFString   [in,out]  style      ''any''
  SFBool     [in,out]  continuous
  XFAny      [in]      start
  XFAny      [in]      finish
...
}
```

Die Felderwerte steuern dabei das Verhalten für eine einzelne Knoteninstanz:

name Definiert den Namen der Aufgabe. Sie wird nur zur Visualisierung eingesetzt, um den Benutzer Auskunft über die aktuelle Aufgabe zu geben.

viewSpace Die 3D-Repräsentation kann entweder an der aktuellen Position des Szenengraphen erscheinen oder skaliert in den sichtbaren Raum der Kamera. Sie erscheint immer vor dem Benutzer. Der Standardwert *TRUE* transfor-

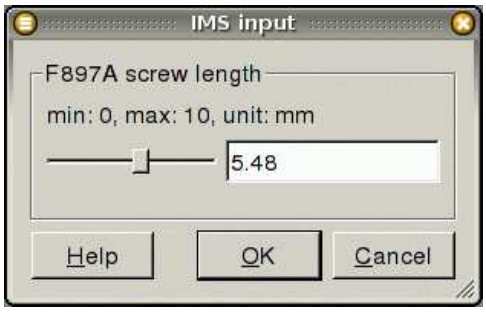

Abbildung 4.15: 2D-Repräsentation einer Quantifizierung

miert die Sensorgeometrie in den Kameraraum. Das hat den Vorteil, dass der Benutzer immer frei navigieren kann.

style Definiert die Repräsentationsform für eine einzelne Instanz. Der Wert "any" übernimmt den aktuellen Wert von der Laufzeitumgebung. Das ist für die meisten Fälle die sinnvollste Belegung. Dennoch ist es möglich, dass man die aktuelle Repräsentation pro Instanz bestimmen kann. Dazu setzt man den *style* Feldwert auf *2D* bzw. *3D*.

continuous bestimmt, ob die Benutzereingaben kontinuierlich versenden werden oder ausschließlich ein finaler Wert.

start Startet die Abarbeitung der Aufgabe und generiert eine graphisch-interaktive Repräsentation.

finish Stoppt die Abarbeitung der Aufgabe und löscht die graphisch-interaktive Repräsentation. Die Repräsentation kann auch eigenständig die Aufgabe abschließen.

Diese Basisfunktionalität erlaubt *TaskSensor*-Knoten unabhängig ihrer eigentlichen Ausprägung zu steuern. Wichtig ist, dass die Entscheidung der Repräsentation, ob 2D (siehe Abbildung 4.15) oder 3D (siehe Abbildung 4.16), vorzugsweise Systemweit für alle Sensorinstanzen gesetzt wird. Diesen globalen Parameter kann der Benutzer für alle Knoten an einer zentralen Stelle beeinflussen.

Im Folgenden werden die einzelnen konkreten Sensor-Typen beschrieben und auf ihre Randbedingungen, sowie eine mögliche Beeinflussung durch andere Komponenten eingegangen:

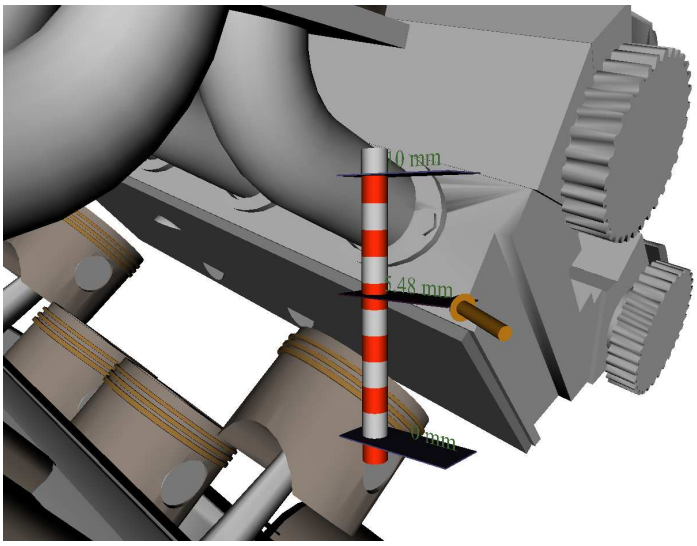

Abbildung 4.16: 3D-Repräsentation einer Quantifizierung

4.4.1 Positionierung

Durch den *PositionTaskSensor* können Objekte im Raum positioniert werden. Dabei unterstützt der Sensor sowohl die freie Positionierung als auch die Positionierung auf einen Gitter.

Intern wird in immersiven Umgebungen der *SpaceSensor* eingesetzt, um interaktiv eine Position mit dem *BodyPart* anzusteuern und auszuwählen. Wichtig ist hierbei, dass man die Genauigkeit der Eingabegeräte berücksichtigt, um die Aufgabe darauf abstimmen zu können.

4.4.2 Selektion

Über den *SelektionTaskSensor* wird eine Teilmenge aus einer gegebenen Menge ausgewählt. Dabei kann der Sensor entweder eine Gruppe von Objekten oder eine Liste von Kommandos anbieten und unterscheidet somit zwischen variabler Selektion (Objekte) oder relativ konstanter Selektionsliste (Kommandos).

Für die Umsetzung in immersiven Umgebungen werden mehrere *TouchSensor* Knoten eingesetzt, die die Selektion eines einzelnen Objektes verarbeiten.

4.4.3 Quantifizierung

Der *QuantizationTaskSensor* erlaubt Werte in einem gegebenen Wertebereich, explizit definiert durch ein Minimum und ein Maximum, einzugeben. Dabei ist wiederum die Genauigkeit der benutzten Eingabegeräte zu beachten die gegeben falls das Ergebnis beeinflussen können.

In immersiven Umgebungen wird ein *PlaneSensor* eingesetzt und so konfiguriert, dass der Benutzter eine 1D Auswahl treffen kann.

4.4.4 Text

Der *TextTaskSensor* erlaubt dem Benutzter direkt einen Text eingeben zu können.

In Immersiven Umgebungen wird dazu der Standard X3D *KeyboardSensor* eingesetzt.

4.4.5 Rotation

In 3D benötigt man neben der Positionierung auch die Rotation, wobei in VR die Rotation als Orientierung im Raum zu verstehen ist. Aspekte der Umsetzung sind die Lage des Rotationszentrums, der Rotationswinkel, Objektentfernung und Größe der Bodypart-Geometry.

Umgesetzt wird diese Aufgabe mit dem *RotationTaskSensor* welcher wiederum in Immersiven Umgebungen einen *SphereSensor* instanziert, um direkt mit dem Benutzter zu interagieren.

4.5 Zusammenfassung

Das im Rahmen dieser Arbeit entwickelte und vorgestellte Konzept zur Interaktion ermöglicht dem Anwendungsentwickler Geräte und Modelle effizient, flexibel und unabhängig von der eigentlich Anwendungsumgebung einzusetzen.

Dabei wurden nicht wie in bisherigen Arbeiten eine einzelne klassifizierte Abstraktion für Gerätegruppen oder Methoden entwickelt, sondern ein dreistufiges Sensor-Konzept vorgestellt, das dem Entwickler ein mächtiges Werkzeug zur Verfügung stellt:

Sensoren für Datenstöme *Data Stream Sensor (DSS)* Objekte erlauben die direkte Anbindung von Ein- und Ausgabeströmen unabhängig von einzelnen Geräten oder Interaktionsaufgaben. Sie sind nicht an Geräteklassen ge-

bunden, sondern abstrahieren einzelne typisierte Datenströme von oder zu Geräten. Daten von physikalischen und virtuellen Geräten werden dabei in die einzelnen Komponenten aufgeteilt und synchronisiert an die Anwendung übergeben.

Sensoren zur direkten Manipulation *Direct Manipulation Sensor (DMS)* Objekte erlauben dem Anwender Teile der Szene graphisch-interaktiv und direkt zu manipulieren. Die Sensoren reagieren auf Veränderungen des Benutzermodells (zum Beispiel der Position der Hand), sind selbst aber unabhängig von konkreten Eingabegeräten. Die Veränderung des Benutzermodells wird direkt über *DSS*-Objekte gesteuert.

Sensoren zur indirekten Manipulation *Indirect Manipulation Sensor* (*IMS*) Objekte ermöglichen der Anwendung Parameter vom Benutzer zu erfragen und definieren somit konkrete Eingabeaufforderungen und Aufgaben. Sie beinhalten aber keine einzelnen graphischen Repräsentationen, sondern wählen selbständig, abhängig von der Laufzeitumgebung, eine geeignete Interaktionsform aus. In immersiven Umgebungen werden zur Interaktion mit den entsprechenden Elementen direkt *DMS* Objekte eingesetzt.

Die drei Sensor-Ebenen bauen direkt aufeinander auf und setzen die Techniken der jeweils tieferen Stufe ein um eine weitere Abstraktion zu schaffen.

Das hier dargestellte Konzept erlaubt dem Anwender Sensoren unabhängig von der Ausprägung der Laufzeitumgebung zu spezifizieren. Sie entsprechen nur abstrakten Interaktionsmodulen die sowohl in Desktop-basierten als auch Immersiven Umgebungen ohne Veränderung gültig sind.

Kapitel 5

Parallelverarbeitung und Skalierbarkeit

Skalierbarkeit eines Softwaresystems bezeichnet die Eigenschaft, unterschiedliche Ausprägungen und Ausbaustufen von Hardware- und/oder Softwareplattformen nutzen zu können. Aspekte der Skalierbarkeit sind die technische Machbarkeit, sowie die vom System erreichte Auslastung bereitgestellter Ressourcen um ein vorgegebenes Ziel zu erreichen. Vorgegebene globale Ziele können dabei zum Beispiel die Anforderung sein, mit einer Frequenz von 30 Hz neue Bilder zu berechnen, oder 50 Schritte pro Sekunde zu simulieren.

5.1 Grundlagen

Grundlage für die Skalierbarkeit eines Gesamtsystems ist die Ausnutzung und Performance der einzelnen Komponente. Die Gesamtperfomance ist wiederum ein sehr wichtiges Kriterium für Virtual-Reality-Systeme und Applikationen im allgemeinen. Die Voraussetzung, auch bei einer zunehmenden Szenenkomplexität die Systemlatenz unter 100 ms zu halten [26], verlangt einen konstant hohen und stabilen Datendurchsatz. Für die Gesamtperformance des Systems sind vor allem vier Parameter entscheidend: die Rechenleistung, die Leistung des Graphiksystems, die Gesamtarchitektur sowie die Möglichkeit der Parallelverarbeitung unterschiedlicher Vorgänge.

5.1.1 Rechenleistung

In der Vergangenheit konnten sich VR-Systementwickler auf die stetig wachsende Leistung von Rechnersystemen verlassen. Die Wachstumsraten haben dabei sogar die Grenzen von Moore's Gesetz [94] durchbrochen: Die CPU Hersteller haben nun alle 18 Monate und nicht wie lange üblich alle 24 Monate die Leistung ihrer Prozessoren verdoppelt.

Nun aber sind erste physikalische Grenzen erreicht [66]. Es ist nicht mehr möglich, die Taktraten von Prozessoren mit vertretbarem Aufwand weiter in diesem Maße zu erhöhen. Aus diesem Grund haben Hardwarehersteller in ihren aktuellen High-End CPUs Mechanismen integriert, die parallel zwei Programmstränge (*Threads*) ausführen können [22]. Die nächsten Generationen werden mehrere komplette Kerne auf einen Chip vereinen [74].

Multiprozessor-Systeme haben eine lange Tradition. Serversysteme mit bis zu mehreren Tausend Prozessoren sind sehr selten, jedoch werden von vielen unterschiedlichen Herstellern Maschinen mit 4 bis 32 Prozessoren angeboten [91][98][49]. Multiprozessor-Workstations sind schon heute gebräuchlich und in Zukunft werden wohl eine Großzahl der PC-Systeme mit mehreren CPUs oder Kernen ausgestattet sein.

Aus diesem Grund ist es wichtig, dass ein aktuelles System möglichst optimal mit einer beliebigen Anzahl von CPUs arbeitet. Dabei ist es entscheidend, wie gut die Aufgaben auf die einzelnen Prozessoren verteilt werden können.

5.1.2 Graphikleistung

Bis Mitte der Neunziger wurden vorwiegend spezielle Graphikmaschinen [93] für VR-Applikationen eingesetzt. Diese Maschinen hatten durch ihre hohe Graphikleistung ein entscheidendes Alleinstellungsmerkmal, was sie zu der bevorzugten Plattform für den wissenschaftlichen und industriellen Einsatz von VR machte. Die Investitionskosten für diese Maschinen war immens hoch, da der Markt relativ klein im Verhältnis zu dem restlichen Hardware-Sektor blieb. Erst Ende der neunziger Jahre begannen die Hersteller von Spielekonsolen simple 3D-Graphiksysteme in ihre Geräte zu integrieren [132]. Durch die Konsolen angespornt, entwickelte sich kurz darauf ein neuer Markt für 3D-Graphiksysteme im PC-Sektor. Diese Graphikkarten wurden, wie die Konsolen, vorwiegend für die Darstellung von Spielen entwickelt und eingesetzt. Der Wettkampf führte zu einer unglaublichen Steigerung der Graphikleistung in den vergangen 10 Jahren. Wenn man sich das

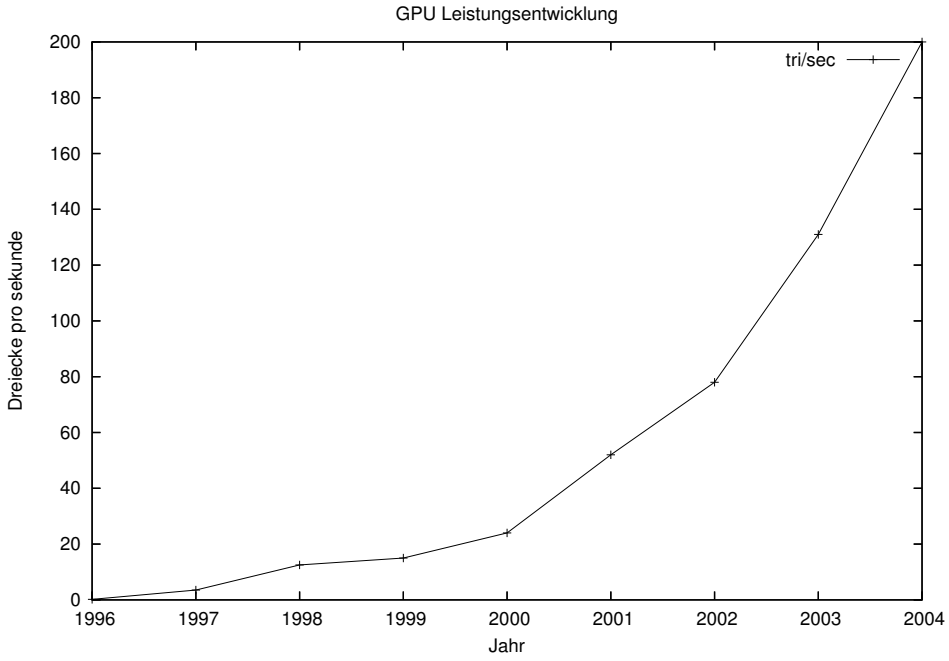

Abbildung 1: Entwicklung der Graphikleistung zwischen 1996 und 2004

Wachstum der maximal in einer Sekunde transformierten und beleuchteten Dreiecke (siehe Abbildung 1) betrachtet, so ergibt sich eine Steigerung um den Faktor 1000: von ungefähr 200.000 Dreiecken 1996 zu 200.000.000 im Jahre 2004.

Diese enorme Leistungssteigerung der graphischen Recheneinheiten (Graphik Processing Unit, GPU) beruht vorwiegend auf verbesserten Algorithmen, höheren Taktraten und vor allem mehr parallelen Verarbeitungseinheiten (Aktuelle Hardware [149] besitzen bis zu 32 Fragment und 16 Vertex-Einheiten). Glücklicherweise übernehmen die Hardwareschnittstellen [118][142] die Parallelisierung. So lange die Applikation genügend Primitives ausreichend schnell liefert, füllen diese Schnittstellen die Einheiten automatisch.

Entscheidend für eine Systemperformance ist es, Verfahren und Methoden anzuwenden die versuchen, die Architekturen möglichst immer an ihrem Limit zu bedienen und somit optimal auszulasten.

.0.3 Gesamtarchitekturen

MR-Applikationen werden auf den unterschiedlichsten Hardware-Systemen implementiert. Vor allem im AR-Umfeld werden in immer mehr wissenschaftli-

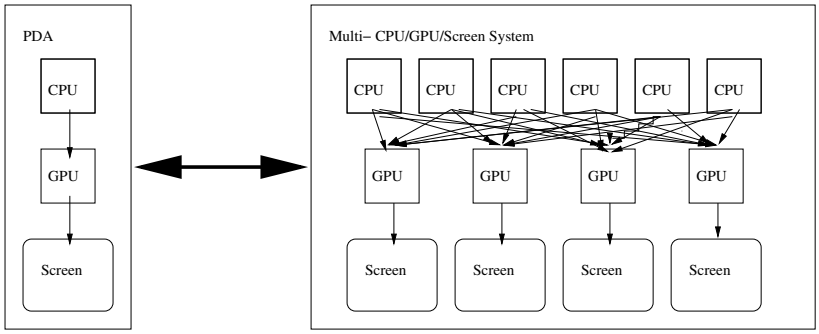

Abbildung 2: Spektrum der MR-Hardwareplattformen

chen und industriellen Anwendungen *Handhelds* oder *Personal Digital Assistance* (PDA) [141][99][86] eingesetzt. Diese Geräte sind leicht, bieten aber auch relativ wenig Rechen- und Graphikleistung. Sie verfügen im Allgemeinen nur über eine CPU, eine simple Graphikeinheit (GPU) und einen Bildschirm mit geringer Auflösung (siehe Abbildung 2). Auf diesen Maschinen ist es nicht sinnvoll mehrere Aufgaben parallel abzuarbeiten, da der zusätzliche Verwaltungsaufwand oft sogar zu Einbußen in der Performance führt. Für den sinnvollen Einsatz solcher Maschinen ist es notwendig, skalierbare Darstellungs- und Animationsalgorithmen zur Anwendung zu bringen, um die geforderten Bild- und Interaktionswiederholraten zu ermöglichen.

Auf der anderen Seite stehen große Multiprozessorsysteme (siehe Abbildung 2), die vorwiegend für klassische immersive VR-Umgebungen mit mehreren Projektionen [31][71][147] eingesetzt werden. Diese Maschinen verfügen über mehrere CPUs und Graphikeinheiten, die wiederum mehrere Bildschirme oder Projektionen betreiben. Die Systemfähigkeit Aufgaben auf diesen Systemen zu Parallelisieren ist essentiell. Dabei sollte das System nicht nur alle CPUs und GPUs auslasten, sondern die Aufgaben auch möglichst optimal verteilen. Somit ist der Einsatz von skalierbaren Algorithmen auch für diese Plattformen sinnvoll.

.1 Parallelverarbeitung

Um von mehreren parallelen Rechenpfaden (*Threads*) zu profitieren, ist es entweder notwendig, vollständig unabhängige Aufgaben mit eigenen Daten zu definieren, oder, bei sehr aufwendigen Aufgaben, in einzelne Pfade aufzuteilen und den Datenzugriff zu synchronisieren. In einer typischen VR-Applikation gibt es beide

Fälle.

Im Allgemeinen gibt es in MR-Anwendungen eine Applikationseinheit, die auf der Grundlage von externen und internen Ereignissen einen neuen Systemstand berechnet und anschließend unterschiedliche Ausgaben (z.B. graphisch oder akustisch) und Evaluierungseinheiten (z.B. Kollisionserkennung) synchronisiert. Diese Ausgaben und Evaluierungseinheiten arbeiten typischerweise auf ihren eigenen Daten und reflektieren nur eine beschränkte Sicht auf den Gesamtzustand. Die einzelnen Verfahren sind dabei gut (z.B. Kollisionserkennung) oder kaum parallelisierbar (z.B. Darstellung auf einer einzelnen Graphikeinheit). Die eigentliche Parallelisierung der Render- und Evaluierungsverfahren ist nicht Gegenstand diese Arbeit, soll aber anhand der Parallelisierung der Ausgabe aufgezeigt werden.

Ein wichtiger Aspekt für das Rahmensystem ist die automatischen Parallelisierung der Applikationseinheit, um die Neuberechnung des Szenenzustands zu beschleunigen.

Grundproblem aller parallelen Systeme ist die Frage der Synchronisation der unterschiedlichen Rechenpfade. Dabei geht es vorwiegend darum zu verhindern, dass mehr als ein Pfad gleichzeitig schreibend oder lesend auf eine Datenstruktur zugreift. Um diese Situation grundsätzlich zu vermeiden, werden hier zwei unterschiedliche Verfahren angewandt:

Für den Applikations- und Verhaltensgraphen werden keine Daten kopiert: die Kanten des Graphen werden benutzt, um unabhängige Rechenpfade zu bestimmen und parallel abzuarbeiten.

Für den kompletten Darstellungsprozess, bestehend aus *culling, state-sorting, rendering* und *clustering,* werden die Daten mit einem gesonderten Darstellungsgraphen synchronisiert.

.1.1 Parallelverarbeitung auf dem Applikationsgraphen

Der Applikationsgraph erlaubt dem Entwickler Verbindungen zwischen Feldern festzulegen und somit ein Model zu spezifizieren, um Nachrichten durch den Verhaltensgraphen zu propagieren. Der Verhaltensgraph besteht aus Knoten und den Feldverbindungen. Zur Auswertung des Graphens werden Nachrichten generiert und in einer definierten Reihenfolge abgearbeitet.

Per Definition ist es den Szenenknoten nicht erlaubt, den Graphen während der Abarbeitung der Nachrichten zu verändern. Es werden keine Kanten oder Knoten als Teil des Verhaltensgraphen gelöscht oder erzeugt. Scriptknoten können den Graphen verändern, müssen dies aber gesondert publizieren. Alle Knoten, die

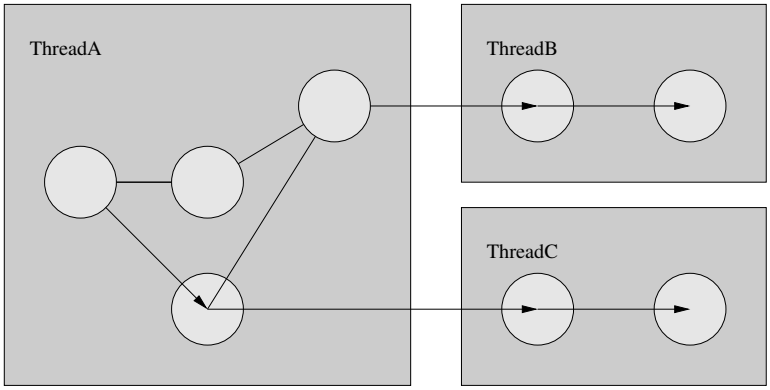

Abbildung 3: Automatische Parallelverarbeitung auf den Event-Pfaden

das Rahmensystem bereitstellt, empfangen Nachrichten. Diese Knoten verändern möglicherweise daraufhin ihren internen Zustand und versenden wiederum Nachrichten. Das System garantiert dieses Verhalten für alle Knoten, ausgenommen für Script-Knoten, die Teil der Szene sind.

Das Rahmensystem benutzt diese Information zur Laufzeit um mehrere Rechenpfade bzw. Nachrichten durch den Graphen parallel zu propagieren. So lange die Pfade isoliert, das heißt azyklisch sind und nur einen Eingang besitzen, startet das System automatisch einen neuen Pfad für jeden Teilgraphen (siehe Abbildung 3).

.1.2 Parallelverarbeitung der Ausgabe

Das System erzeugt und verwaltet den Szenengraph für alle applikationsspezifischen und dynamischen Aspekte. Es erlaubt Knoten an unterschiedlichen Stellen im Graphen wiederzuverwenden. Ein einzelner Knoten kann somit auf mehrere Väter verweisen. Dies erleichtert die Applikationsentwicklung, aber verhindert, dass das System nur eine einzige Transformation oder ein umschließendes Volumen (*Bounding Volume*) pro Knoten anlegen kann.

Graphen mit nur einem Vater (*single-parent*) und einer einzigen Transformation pro Knoten sind besser geeignet für die Umsetzung von Darstellungsverfahren. Aus diesem Grund generiert und synchronisiert das System einen *single-parent* Graphen auf Basis von OpenSG[105]. Dabei werden aber nicht generell alle Knoten und Felder kopiert, sondern nur die Knoten, die zur Visualisierung benötigt werden, *Multi-Field* Daten werden vorzugsweise referenziert (siehe Abbildung 4).

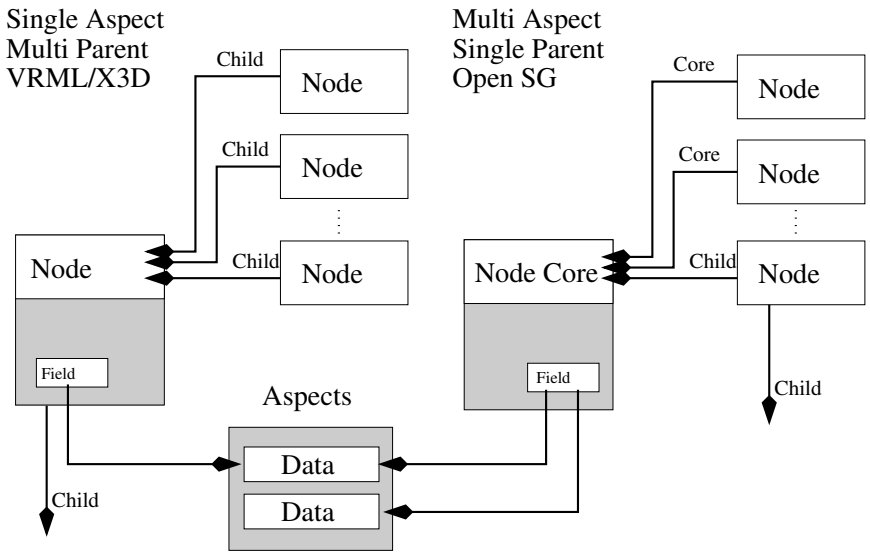

Abbildung 4: Applikations- und Darstellungsgraph

Dieser zweite Graph wird auch benutzt um alle darstellungsspezifischen Optimierungen wie *state-sorting* und *culling* durchzuführen.

Ein weiterer entscheidender Vorteil dieses Designs ist die Möglichkeit, auf den Darstellungsgraphen parallel zu arbeiten. Dazu ist der zweite Graph zwingend notwendig, da es dem Applikationsthread erlaubt ist, jederzeit Feldwerte oder sogar Teile des Graphen zu verändern.

Der Darstellungsgraph besitzt somit seine eigene Sicht auf die Daten aber nicht zwingend eine einfache Kopie. Vor jeder Traversierung des Darstellungsgraphen werden diese unterschiedlichen Sichten synchronisiert. Diese Synchronisation kann auch über Netzwerke erfolgen, um mehrere verteile Darstellungseinheiten (*cluster*) zu unterstützen [113].

.2 Skalierung der Applikations- und Darstellungsleistung

Durch die Parallelisierung von Darstellung und Anwendung ergibt sich ein neues Ablaufmodell. Das vereinfachte Modell, dass in 3.1 vorgestellt wurde, geht davon aus, dass in jedem Zyklus Anwendung und Darstellung sequentiell nacheinander abgearbeitet werden. Durch die Parallelisierung wird nun die Veränderung der Szene in Zyklus $n+1$ berechnet, während parallel der Zyklus n zur Darstellung kommt (siehe Abbildung **5).**

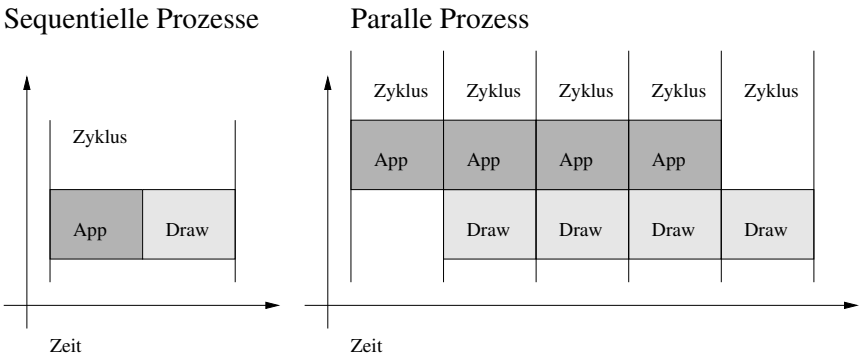

Abbildung 5: Sequentielle und parallele Verarbeitung von Zyklen

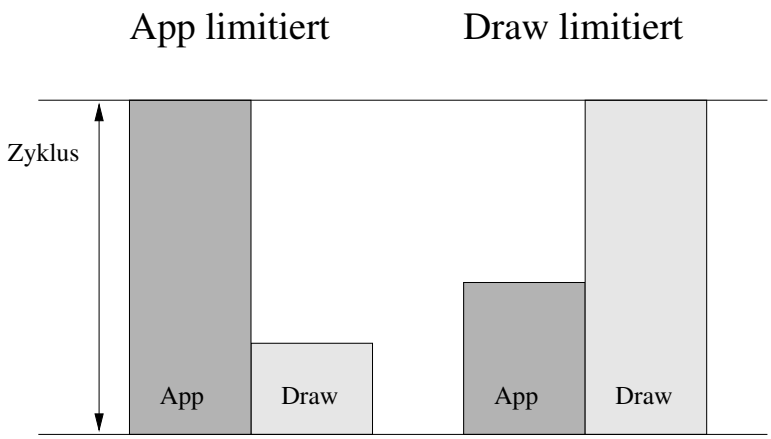

Abbildung 6: Applikations- und darstellungslimitierte Zyklen

Die Zyklen werden immer noch synchron abgearbeitet. Am Synchronisationspunkt muss entweder der Applikationsprozess oder der Darstellungsprozess warten. Daraus ergeben sich zwei Klassen von Applikationen (siehe Abbildung 6).

Applikationslimitiert Die Neuberechnung des Szenenzustands benötigt mehr Zeit als deren Darstellung. Dies ist vor allem bei komplexen Simulationen und Animation (zum Beispiel bei aufwendigen Partikelsimulationen) der Fall.

Darstellungslimitiert Die Darstellung der Anwendung beansprucht mehr Zeit als die Neuberechnung der Szene. Dies ist vor allem bei klassischen VR-Applikationen der Fall, die relativ statisch sind (zum Beispiel die virtuelle Begehung eines statischen Gebäudes).

Diese statische Verteilung ist nicht sinnvoll, da bei Mehrprozessorsystemen Ressourcen ungenutzt brachliegen. Viel sinnvoller ist der Einsatz von skalierbaren Simulations-, Animations- und Darstellungsverfahren. Dabei wird ein einheitliches Modell verfolgt, das einen einzelnen Skalierungsfaktor pro Teilaufgabe bzw. Knoten verwendet. Diese Auflösung wird zur Laufzeit wie folgt bestimmt:

$$R_r = R_n * R_d * R_p \tag{1}$$

R_r Die resultierende Auflösung pro Knoten.

R_n Der Skalierungsfaktor pro Knoten. Damit ist es möglich, unterschiedliche Instanzen gesondert zu gewichten.

R_d Der entfernungsabhängige Anteil. Mit diesem Faktor wird die Skalierung abhängig von der Distanz zwischen Kamera und Objekt skaliert. Der Wert 1 bedeutet dabei, dass Kamera- und Objektposition übereinstimmen. Der Wert 0 bedeutet, dass das Objekt außerhalb eines systemweiten Grenzwertes liegt, der im allgemeinen mit der *back-clipping-plan* übereinstimmt.

R_p Der Skalierungsfaktor der für einen gesamten Prozess steht. Somit können alle beteiligten Knoten in einem Prozess skaliert werden.

Daraus ergibt sich die Forderung, dass das System einzelne Methoden und Verfahren für unterschiedliche Prozesse auf Knotenbasis bereitstellen muss, deren Kosten zumindest näherungsweise skalierbar sein sollten. Exemplarisch werden in den folgenden Abschnitten unterschiedliche Verfahren vorgestellt, die für den Anwendungs- und Darstellungsprozess diese Voraussetzung erfüllen.

Die Skalierbarkeit der Verfahren kann und wird aber nicht nur dazu benutzt die Ressourcen der einzelnen Prozesse besser auszunutzen, sondern auch für die dynamische Bestimmung der Laufzeiten von kompletten Applikationszyklen.

.3 Skalierbare Verfahren zur Darstellung

In diesem Abschnitt werden exemplarisch zwei skalierbare Verfahren zur Ausgabe vorgestellt. Dabei werden deren Eigenschaften untersucht und wesentliche Verbesserungen eingeführt. Die Integration in das Rahmensystem wird diskutiert und die Laufzeiteigenschaften bewertet.

.3.1 Progressive Darstellung von Polygonnetzen

Dreiecks- und Polygonnetze sind in den heutigen VR/AR-Systemen die bevorzugte Objektbeschreibungsform. Das liegt zum einen daran, dass heutige GPU-Architekturen auf die Verarbeitung von Polygonnetzen optimiert sind und zum anderen daran, dass eine Vielzahl von Verfahren und interaktiven Modellierungstools vorzugsweise Polygondaten erzeugen.

Die einzelnen Polygonmodelle besitzen im allgemeinen einen sehr hohen Detaillierungsgrad, um die wachsenden Ansprüche an realitätsgetreuer Darstellung gerecht zu werden. Die Darstellung der einzelnen Dreiecke ist ein nahezu lineares Problem, abgesehen von der Füll- und Geometrielimitierung. Nimmt die Anzahl der Dreiecke zu, steigt der dafür benötigte Zeitaufwand in etwa um den gleichen Faktor.

Um den Aufwand zu kontrollieren ist es notwendig, die Anzahl der Dreiecke zu reduzieren. Dabei ist es wünschenswert Verfahren zu entwickeln, die die Anzahl der dargestellten Dreiecke zur Laufzeit festlegen, um dementsprechend den Aufwand zu bestimmen.

Progessive- oder Multiresolution-Netze sind Datenstrukturen, die es erlauben eine beliebige Auflösungsstufe aus einem Grundnetz zur Laufzeit abzuleiten.

.3.1.1 Grundlagen

Progressive Netze haben eine Vielzahl von Anwendungsgebieten. Neben der statischen und dynamischen Reduktion von Polygonnetzen, findet vor allem die progressive Übertragung und die Kompression von Netzen häufig Anwendung.

Es gibt eine große Anzahl an Polygon-Reduktionsverfahren; eine gute Übersicht bietet [43]. Die modernen progressiven Verfahren bauen fast alle auf den Ideen und Datenstrukturen auf, die Hoppe 1997 [51] erstmals veröffentlich hat. Alle arbeiten auf Dreiecksnetzen (siehe Abschnitt .3.1.1.1), definieren Operatoren auf den Netzprimitiven (siehe Abschnitt .3.1.1.2) und definieren eine Metrik zur Auswahl von geeigneten Primitiven. Variationen wie View-Dependent [52] oder Out-of-core [78] Algorithmen werden hier nicht betrachtet, da sie für den Anwendungsfall und heutige GPU-Architekturen nur bedingt relevant sind.

.3.1.1.1 Polygon- und Dreiecksnetze Oberflächen in der Computer-Graphik werden oft als Dreiecksnetz repräsentiert. Netze, die komplexe Polygone wie n-seitige Flächen oder Flächen mit Löchern beinhalten, sind einfach und verlustfrei in

Dreiecksnetze zu überführen. Um die Datenstruktur einfach und effizient zu halten, beschränken sich die meisten Verfahren zur Reduktion auf Dreiecke.

Geometrisch ist ein Dreiecksnetz eine stückweise lineare Oberfläche, die aus dreieckigen Flächen besteht, die an ihren Kanten zusammengefügt sind. Hoppe [54] definiert eine Netz-Geometrie als Tupel (K, V), wobei K die Konnektivität des Netzes (die Angrenzung der Eckpunkte, Kanten und Fläche) $V = \{v_1, ..., v_m\}$ die Menge der Eckpunktpositionen ist welche die Form des Körpers im $\mathbb{R}^3$ bestimmen.

In den meisten Fällen sind zusätzlich Attribute neben der Geometrie mit dem Netz verbunden. Diese Attribute können in zwei Kategorien aufgeteilt werden: Diskrete und skalare Attribute.

Diskrete Attribute Diese Werte sind im allgemeinen mit Flächen verbunden. Ein diskretes Attribut ist zum Beispiel das *Material*. Das *Material* definiert die *Shader*-Funktion und die Parameter zur Darstellung der Fläche. Eine einfache Shader-Funktion ist zum Beispiel eine zugewiesene Textur.

Skalare Attribute Diese Attribute sind im allgemeinen mit der Geometrie verbunden und definieren zum Beispiel Farben ($color(r, g, b)$), Normalen ($normal(n_x, n_y, n_z)$) und Texturkoordinaten ($texCoord(u, v)$). Diese Werte spezifizieren einen lokalen Parameter für die Shader-Funktion. In einfachen Fällen sind sie direkt mit Eckpunkten verbunden. Jedoch ist es notwendig, um die Unstetigkeiten in dem Skalarfeld abbilden zu können, die Skalarwerte nicht mit den Eckpunkten, sondern mit den Ecken eines Netzes zu assoziieren [5]. Eine Ecke ($corner$) ist dabei als ein Eckpunkt/Fläche-Tupel definiert. Skalare Attribute an einer Ecke $corner(v, f)$ spezifizieren dabei die Shader-Parameter für die Fläche f am Eckpunkt v.

Somit ist das Netz das Tupel $M = (K, V, D, S)$, wobei V die Geometrie bestimmt, D die Menge der diskreten Attribute d_f, welche mit den Flächen $f = \{j, k, l\} \in K$ assoziiert sind, und S die Menge der skalaren Attribute $s_{(v,f)}$ verbunden mit den Ecken $corner(v, f)$.

Die Attribute D und S führen zu Brüchen in der visuellen Erscheinung der Oberfläche. Eine Kante $edge\{v_j, v_k\}$nennt man dabei *Scharfe-Kante,* wenn sie eine der Folgenden Eigenschaften erfüllt:

1. Die zwei benachbarten Dreiecke f_l und f_r haben unterschiedliche diskrete Attribute (z.b. $d_{f_l} \neq d_{f_r}$).

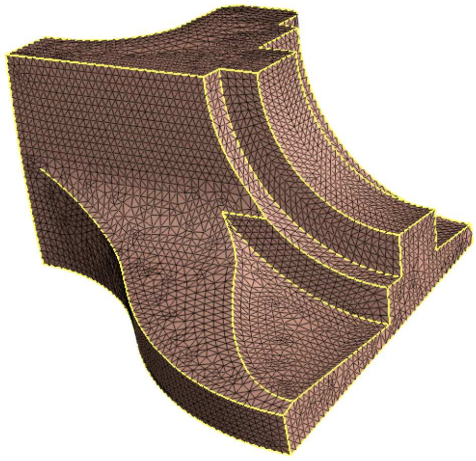

Abbildung 7: Scharfe Kanten durch unterschiedliche skalare Attribute

2. Die angrenzenden Ecken haben unterschiedliche skalare Attribute (z.b. $s_{(v_j,f_l)} \neq s_{(v_j,f_r)}$).

3. Sie definiert eine Kante, besitzt nur ein benachbartes Dreieck.

Fast alle Szenengraph-Systeme erlauben nur ein diskretes Attribut pro Geometrieknoten, da es sonst nicht ausreicht, nur auf Knotenebene nach Materialen zu sortieren. Somit ist der 1. Fall gegeben, Geometrieknoten als Element eines Szenengrahen auszuschließen. Die beiden anderen Fälle sind jedoch wichtige Kriterien und finden auch in Geometrieknoten Anwendung (siehe die gelben Linien in Abbildung 7).

.3.1.1.2 Progressive Netz-Repräsentation Zur Erstellung der reduzierten Auflösungen ist es notwendig, geeignete Transitionen auf dem Netz zu definieren, um Elemente zu entfernen. Hoppe definiert hierzu zunächst drei Transitionen [54]: Kanten kollabieren, Kanten aufschneiden und Kanten vertauschen. In einer späteren Veröffentlichung [51] zeigt er, dass es ausreicht, nur eine einzelne Transition zu definieren, um ein Netz effizient zu verfeinern.

Das initiale Netz wird durch wiederholte Anwendung einer Transition *ecol* (Kantenkollabierung) reduziert. Dabei wird eine Kante (v_s, v_t) zu einem neuen Knoten v_s kollabiert (siehe Abbildung 8). Die maximal zwei Dreiecke, die diese Kante enthalten, werden entfernt. Die neue Position von v_s sollte so gewählt

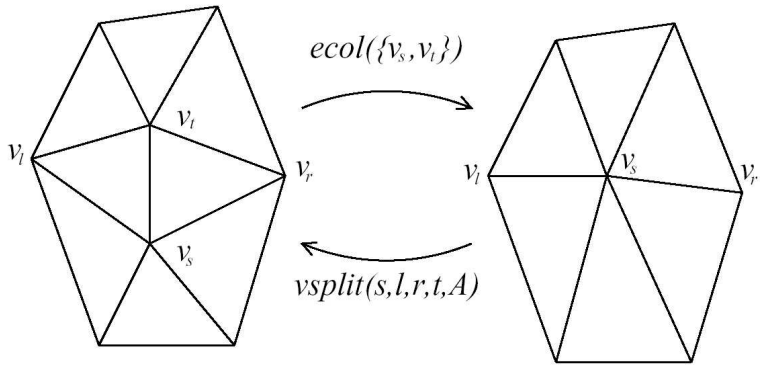

Abbildung 8: ecol- und vsplit Transformation

werden, dass das neue Netz optimal die Form des initialen Netzes wiedergibt. Außerdem müssen die diskreten und skalaren Attribute im Netz aktualisiert werden.

Durch mehrfache Anwendung der *ecol*-Transition wird eine Sequenz aus Netzen mit immer weniger Dreiecken erzeugt. So wird das initiale Netz $\hat{M} = M^n$ rekursiv in ein gröberes Netz M^0überführt:

$$(\hat{M} = M^n) - ecol_{n-1} \rightarrow ... - ecol_1 \rightarrow M^1 - ecol_0 \rightarrow M^0$$

Die ecol-Transition ist invertierbar. Die Umkehrfunktion heißt *vsplit* (Knoten-Spaltung), sie fügt einen neuen Knoten v_t in der Nähe von v_s hinzu. Außerdem werden die Kanten umgeordnet, so dass die Dreiecke $\{v_t, v_s, v_l\}$ und $\{v_t, v_s, v_r\}$ wieder im Netz vorhanden sind (siehe Abbildung 8). Die Position von v_s und alle Attribute werden aktualisiert. Ein feines Netz M^0 wird schrittweise vergröbert um final das Netz M^nzu erhalten:

$$M^0 - vsplit_0 \rightarrow M^1 - vsplit_1 \rightarrow ... - vsplit_{n-1} \rightarrow (\hat{M} = M^n)$$

$M^0\{vsplit_0, ..., vsplit_{n-1}\}$ nennt man eine progressive Netz-Repräsentation.

.3.1.1.3 Netz Simplifizierung Die Qualität der resultierenden Simplifizierung ist im wesentlichen abhängig von zwei Parametern:

1. Die Auswahl der nächsten zu kollabierenden Kante.

2. Die Bestimmung der neuen Position für v_s.

Für diese Teilaufgaben wurden unterschiedliche Lösungen in der Literatur entwickelt. Detaillierte Übersichten und Gegenüberstellungen sind in [43] und [126] zu finden.

Einige wichtige Verfahren werden hier kurz vorgestellt und bewertet.

.3.1.1.4 Explizite Energiefunktionen Hoppe et al [54, 51] definiert eine Explizite Energiefunktion. Zur Bestimmung der günstigsten Kante wird dabei eine Energiefunktion verwendet, die Aufschluss darüber gibt, wie stark das Modell K durch die Kollabierung dieser Kante verändert werden würde.

$$\triangle E = E_{K'} - E_K$$

Die Energiefunktion wird wie folgt zusammengesetzt:

$$E(M) = E_{dist}(M) + E_{sprint}(M) + E_{scalar}(M) + E_{disc}(M)$$

$E_{dist}(M)$ ist dabei die quadratische Abweichung vom Netz, $E_{spring}(M)$ beschreibt die Verzerrung der Geometrie, $E_{scalar}(M)$ die Abweichung der skalaren Attribute und $E_{disc}(M)$ bestraft die Veränderung von Diskontinuitäten.

Die Position von v_s wird wie folgt berechnet: $v_s^i = (1-\alpha)v_s^{i+1} + v_t^{i+1}$für $\alpha = \{0, \frac{1}{2}, 1\}$ und wird dann so gewählt, dass das neuen Netz optimal die Form des initialen Netzes wiedergibt.

Die Schritte zur Simplifizierung sind im einzelnen:

1. Für eine Netztransformation $K \rightarrow K'$ wird $\triangle E$ für jede Kante bestimmt und sortiert.

2. Iterativ wird die Kante mit den günstigsten $\triangle E$ entfernt, v_s und die Energiekosten der benachbarten Kanten neu berechnet und wieder einsortiert.

3. Abbruch beim Erreichen der gewünschten Flächenanzahl.

.3.1.1.5 Fehlerquadriken In [44] beschreiben Paul S. Heckbert und Michael Garland einen Ansatz, der auf Fehlerquadriken pro Eckpunkt aufbaut und erweitern das Verfahren ein Jahr später so, dass es auch Skalare und Diskrete Attribute wie Farben und TextureCoordinate berücksichtigt[45]. In [55] wird ebenfalls eine Erweiterung zur Handhabe von Punkt-Attributen vorgestellt und in [46] das Verfahren auf beliebige Dimensionen erweitert.

Die Metrik selbst basiert auf einer Quadrik, die jedem Knoten im Netz zugewiesen wird. Diese Quadrik misst den Abstand zwischen den Knoten und der Menge der angeschlossenen Flächen. Dazu wird ein Knoten als Schnittpunkt einer Menge von Dreiecken betrachtet, die den Knoten als Eckpunkt besitzen. Die Metrik wird dann als die Summe der quadratischen Abstände des Knotens zu diesen Ebenen definiert:

$$\triangle(v) = \triangle([v_x v_y v_z 1]^T) = \sum_{p \in planes(v)} (p^T v)^2$$

Wobei p = $[abcd]^T$ durch $ax + by + cx + d = 0$ und $a^2 + b^2 + c^2 = 1$ definiert ist. Durch Umformung erhält man:

$$\triangle(v) = \sum_{p \in planes(v)} (v^T p)(p^T v) = \sum_{p \in planes(v)} v^T (pp^T) v = v^T (\sum_{p \in planes(v)} K_p) v$$

$K_p$wird als fundamentale Fehlerquadrik bezeichnet:

$$K_p = pp^T = \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{pmatrix}$$

Mit Q wird dabei die Summe dieser K_p bezeichnet und heißt Fehlerquadrik:

$$Q = \sum_{p \in planes(v)} K_p$$

Somit wird die gesamte Menge die zu v inzidenten Dreiecksebenen durch eine einzelne Matrix Q repräsentiert. Bei einer Paarkontraktion v_1, v_2 wird anstelle der Vereinigung der Ebenen die Summe der Quadriken $Q_1 + Q_2$ berechnet. Diese Summe entspricht einer Vereinigung, wenn die Flächen von Q_1 und Q_2 disjunkt sind. Bei einer Überlappung (das passiert, wenn $v_1$und v_2 auf einer Kante liegen) wird ein Dreieck höchstens drei mal mitgezählt. Dadurch wird zwar die Fehlergröße ungenau, jedoch ist die Matrizenrechnung effizienter und benötigt weniger Speicher als die Mengenvereinigung.

Beim Start des Algorithmus ist der Wert von Q für jeden Knoten gleich 0, da der Knoten im Schnittpunkt der Ebenen liegt. Bei der Kontraktion eines Paares werden die Q-Matrizen der beiden ursprünglichen Knoten summiert und bilden

eine neue Fehlerquadrik für den neuen Knoten. Damit wird der Fehler, der durch die Vereinigung der beiden Knoten entstanden ist, beschrieben.

Für die Bestimmung der neuen Position v_s wird die Fehlerfunktion abgeleitet und zu Null gesetzt. Das ist äquivalent zu:

$$\begin{pmatrix} q11 & q12 & q13 & q14 \\ q12 & q22 & q23 & q24 \\ q13 & q23 & q33 & q34 \\ 0 & 0 & 0 & 1 \end{pmatrix} v_s = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Ist die Matrix invertierbar, so erhält man folgenden Gleichung:

$$v_s = \begin{pmatrix} q11 & q12 & q13 & q14 \\ q12 & q22 & q23 & q24 \\ q13 & q23 & q33 & q34 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Sollte die Matrix nicht invertierbar sein, wird v_s wie in .3.1.1.4 bestimmt.
Der Algorithmus besteht in den einzelnen Schritten aus:

1. Berechnung der Q-Matrizen für alle Knoten.
2. Auswahl aller gültigen Paare.
3. Berechnung des Zusammenfassungsziels v_s für jedes Paar (v_1, v_2). Der Fehler $v_s^T(Q1 + Q2)v_s$ stellt die Kosten der Zusammenfassung dar.
4. Paare sortieren.
5. Paar mit geringsten Kosten entfernen.

.3.1.1.6 Lindstrom und Turk Peter Lindstrom und Greg Turk beschreiben in [79] einen Algorithmus, der eine hochqualitative Annäherung an das Originalmodell erreicht, ohne dabei während des Reduktionsprozesses Informationen über das anfängliche Modell zu speichern. Die Kosten werden wie folgt berechnet:

$$f_C(e, v) = \lambda * f_v(e, v) + (1 - \lambda) * ||e||^2 * f_B(e, v)$$

e ist die Kante, die geschrumpft werden soll, v der Zielpunkt der Schrumpfung, $\lambda = \frac{1}{2}$, $f_v(e, v)$ die Funktion zur Volumenoptimierung (siehe Formel 2) und

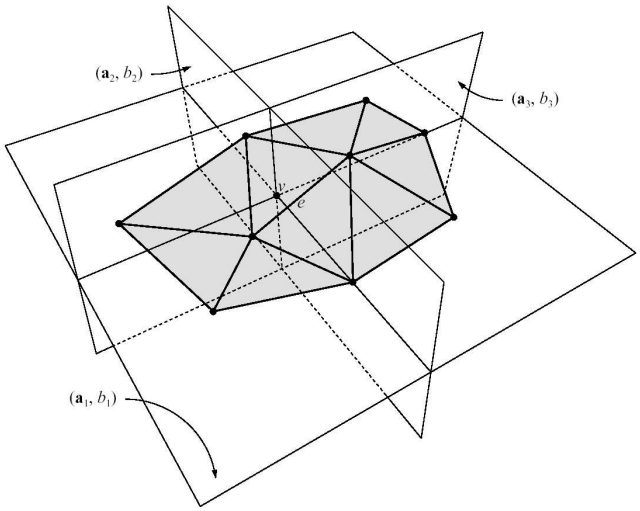

Abbildung 9: Durch drei Optimierungsebene definierte Schnittpunkt

$f_B(e, v)$ die Funktion zur Randoptimung (siehe Formel 3).

Die Berechnung des optimalen Zielpunktes wird als Optimierungsvorgang betrachtet. Zuerst wird versucht, einen Zielpunkt zu errechnen, so dass das Volumen eines geschlossenen Modells bewahrt wird. Liegt dieser Eckpunkt in der Nähe der Aussenkante des Modells wird ebenfalls versucht, die Größe der anliegenden Fläche zu bewahren. Da diese beiden Bedingungen meist noch nicht zur eindeutigen Bestimmung dieses Zielpunktes ausreichen, werden weitere geometrische Eigenschaften einbezogen. Jede dieser Bedingungen beschreibt eine Ebene. Hat man 3 Ebenen, die voneinander unabhängig sind, beschreibt deren Schnittpunkt den optimalen Zielpunkt der Kantenschrumpfung (siehe Abbildung 9). Sollte die hierbei bestimmte Lösung noch nicht eindeutig sein, wird zusätzlich ein weiteres Kriterium zur Optimierung des Aussehens der Dreiecke benutzt, welches die Vermeidung von langen, dünnen Dreiecken als Ziel hat (siehe Formel 4). Dabei werden folgende Formeln benutzt:

$$f_v(e, v) = \sum_i V(v, v_0^{t_i}, v_1^{t_i}, v_2^{t_i})^2 \tag{2}$$

$$f_B(e, v) = \sum_i A(v, v_0^{e_i}, v_1^{e_i})^2 \tag{3}$$

$$f_S(e,v) = \sum_i ||(v - v_i)||^2 \tag{4}$$

Wobei t_i das i-te Dreieck, die Kante e, e_i die i-te Aussenkante der Kante, e, $v_i$der i-te Eckpunkt zur Kante e, v_x^y der Eckpunkt x des Primitives y.

Sehr positiv zu bewerten ist die Möglichkeit, große Modelle aufgrund des niedrigen Speicherverbrauchs reduzieren zu können. Das Laufzeitverhalten und die Ergebnismodelle sind ebenfalls gut.

.3.1.1.7 Melax Der von Stan Melax in [88] vorgestellte Algorithmus benötigt ebenfalls wie der Algorithmus von Lindstrom und Turk (siehe Abschnitt .3.1.1.6) keine Referenz zum Originalmodell während der Reduktion. Im Gegensatz zu den bisher vorgestellten Methoden spezialisiert sich dieser Algorithmus auf das Erzeugen von Modellen mit sehr niedriger Polygonzahl innerhalb sehr kurzer Zeit. Interessant ist vor allem, dass er keine neuen Punkte $v_s$erzeugt, sondern immer die Kante auf v_1 schrumpft.

Um die Kosten einer Kantenschrumpfung zu bestimmen, wird die folgende Formel benutzt, die sowohl die Länge dieser Kante als auch die Krümmung der angrenzenden Dreiecke mit in Betracht zieht:

$$c(v_0, v_1) = ||v_0 - v_1|| * \max_{f \in T_{v_0}} \left\{ \min_{n \in T_{v_0 v_1}} \left(\frac{1 - f.normal - n.normal}{2} \right) \right\}$$

Wobei T_{v_0} die Menge aller Dreicke an v_0 und $T_{v_0 v_1}$ alle Dreiecke an der Kante (v_0, v_1) sind. Zu beachten ist, dass der Algorithmus unterschiedliche Kosten für $c(v_0, v_1)$ und $c(v_1, v_0)$ errechnet und somit auch nicht mehr v_0 und v_1 näher untersucht werden müssen, um v_s zu bestimmen.

Der Algorithmus ist sehr schnell, kann aber nicht mit den Ergebnissen der voher gennanten Verfahren mithalten. Er ist aber sehr einfach zu implementieren, hat ein sehr gutes Laufzeitverhalten und erlaubt eine sehr effiziente Darstellung, da keine Punktkoordinaten verändert werden müssen.

.3.1.2 Kodierung von Progressiven Netzen

Es wurden unterschiedliche Modelle entwickelt um Progressive Netze, abhängig vom Anwendungsgebiet, effizient zu kodieren. Dabei sind folgenden Voraussetzungen zu unterscheiden:

Speichereffizienz Der Speicherverbrauch der Datenstruktur, der in statische und dynamische Anteile pro Instanz aufzuteilen ist.

Progressivität Sind die Strukturen nur geeignet um effizient Multiresolution-Netze aufzubauen, oder können sie auch für das progressive Übertragen und Verarbeiten genutzt werden?

Laufzeitverhalten Wie aufwendig sind Veränderungen in der Netzstruktur? Vor allem ist die Auswahl einer bestimmten Auflösungsstufe zu betrachten sowie die edge-collaps und vertex-split Operatoren.

Darstellungseffizienz Wie schnell und gut sind die Strukturen mit aktuellen GPU-Systemen umsetzbar? Interessant ist, wie gut die Verfahren vertex-cache Strukturen nutzen.

In [53] wurden die ersten speziellen Strukturen entwickelt und vorgestellt, die es erlauben, Progressive Netze effizient im Speicher abzulegen und zu traversieren. Diese Datenstruktur kodiert explizit M^0 und eine Liste von *Vertex-Split*-Operationen, die zur Laufzeit effizient abgearbeitet werden können, sich aber nur bedingt für die Darstellung eignen, da sie keine Rücksicht auf das vertex-cache Verhalten nehmen. Die Halb-Kanten-Datenstuktur [23] ist eine Verallgemeinerung dieser Struktur, die jedoch nur bedingt zur effizienten Darstellung geeignet ist, aber weit mehr Operatoren auf dem Netz bereitstellt.

Methoden zur Generierung von vertex-cache freundlichen Strukturen [11][108] sind sehr effizient darstellbar, aber bieten im allgemeinen nur schlechte Unterstützung für Multiresolution Strukturen.

Spezielle Datenstrukturen, die versuchen sowohl Anforderungen an die Verarbeitungs- als auch Darstellungseffizienz gerecht zu werden [37][15][119], verbrauchen durch ihre Komplexität wesentlich mehr Speicher.

Für den in dieser Arbeit vorgestellten Framework wurden die Ergebnisse aus [108] so erweitert, dass es auch edge-collaps und vertex-split Operationen auf dem Netz unterstützt. Es wird aber keine spezifische “Skip-List” [37] erzeugt, sondern das frühe und schnelle Verwerfen von ungültigen Dreiecken auf der Hardware ausgenutzt. Dadurch wird wesentlich weniger Speicher benötigt.

Da genauso wie in [37] eine statische Liste von optimierten *Triangle-Strips* [97] für M^n generiert wird, nutzt das Verfahren vertex-cache Hardware für hohe Auflösungsstufen sehr gut aus. Das Problem ist, dass es mit jedem edge-collaps bis

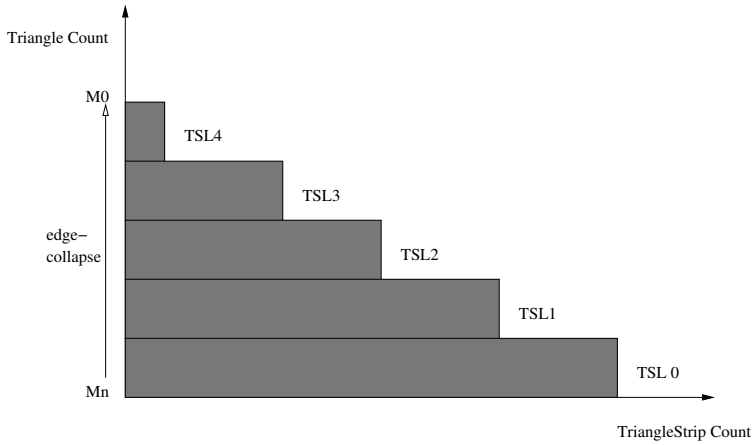

Abbildung 10: TriangleStrip Pyramide zur Optimierung der Vertex-Cache Auslastung

zu zwei *Triangle-Strips* zerstört, und somit sich das Cache-Verhalten verschlechtert. Dieser Umstand ist vor allem unangenehm, da das Verfahren eigentlich vor allem niedrige Level mit wenigen Dreiecken besonders effizient darstellen sollte.

Aus diesem Grund wird an dieser Stelle eine Verbesserung eingeführt: Dazu wird eine Auflösungspyramide von *Triangle-Strip* Listen erstellt. Während der Darstellung wird nicht wie bisher die *Triangle-Strip*-Liste (TSL) 0 für M^n benutzt, sondern ausgehend von der aktuellen Dreiecksanzahl m der nächste Level gewählt, der mehr als m Dreiecke beinhaltet (siehe Abbildung 10). Der Algorithmus zur Erstellung der Pyramide kann folgendermaßen veranschaulicht werden:

```
Halb-Kanten Datenstruktur für M^nerstellen
Triangle-Strip Liste fuer Level 0 erstellen
nextLevel = n / levelScale;
While (M^0 noch nicht erreicht) {
  if (triCount == nextLevel) {
    TriangleStrip Liste erzeugen und speichern
    nextLevel = nextLevel / levelScale
  }
  nächste Kante für edge-collaps bestimmen
  edge-collaps durchführen
  vertex-map in die Liste eintragen.
}
```

Durch diese Erweiterung erhält man ein Verfahren, das vertex-cache Strukturen sehr gut ausnutzt, aber dennoch relativ wenig Speicher verbraucht. Ein entscheidender Vorteil dieses Verfahrens ist die Skalierbarkeit der Darstellungskosten gegenüber dem aktuellen Speicherverbrauch. Benutzt man eine Pyramide mit wenigen TSL-Ebenen und reduziert somit den Speicherverbrauch, dann reduziert sich auch die Renderleistung. Erhöht man die Anzahl der Ebenen, nutzt man das Verfahren der vertex-cache Strukturen weitaus effizienter.

.3.1.3 Integration

In diesem Abschnitt werden auf der Grundlage der X3D-Spezifikation [143] Erweiterungen entwickelt mit dem Ziel, Multiresolution Polygonnetze in das Rahmensystem zu integrieren.

In X3D werden im allgemeinen Polygonnetzobjekte als *IndexedFaceSet* (IFS) [143] Knoten kodiert. IFS Knoten haben selbst nur einen Darstellungsparameter (z.B. *solid* für ein- oder zweiseitiges Polygone), Index-Felder aber keine skalaren Punkt-Attribute. Die Punkt-Attribute wie Position, Normale, Farbe und Texturkoordinaten stehen in referenzierten Kind-Knoten.

Zur Integration der Multiresolution Netze wird kein eigener neuer Knotentyp erzeugt, sondern der IFS Knotentyp um drei Felder erweitert:

```
IndexedFaceSet : Geometry {
...
  SFFloat     1.0     resoltion
  MFInt32     []      collapseMap TRUE
  SFString    [auto]  multiResGen
  SFString    [auto]  multiResViz
...
}
```

Die Parameter sind im Einzelnen:

resolution Definiert die R_n objektabhängige Skalierung der Auflösung (siehe Formel 1).

collapseMap Stellt eine Liste von vorberechneten edge-collapse Operationen bereit. Jede Operation benötigt drei Integer-Werte: Der 1. und 2. Parameter beschreiben die Punkte die zu kollabierte sind. Der 3. Wert wird als Index

in einem regulären Gitter benutzt, welches den Raum zwischen den zwei Punkten quantifiziert.

multiResGen Steuert die Generierung der Multiresolution Struktur.

- **none** Erzeugt kein progressives Netz
- **nice** Erzeugt ein progressives Netz mit Hilfe von Fehlerquadriken (siehe Abschnitt .3.1.1.5). Für jeden kollabierte Kante wird eine neue v_s Position bestimmt.
- **fast** Erzeugt ein progressives Netz mit Hilfe von der simplen Metrik von Melax (siehe Abschnitt .3.1.1.7). Es wird keine neue Position für v_s bestimmt, sondern entweder v_1 oder v_2 benutzt.
- **auto** Der Knoten liest den multiRedGen Wert nicht aus dem Feld, sondern aus dem aktuellen Kontext. Somit ist es möglich, einen Standardwert im Kontext zu setzen, der von allen Knoten mit *auto* Feldwert während der Initialisierung übernommen wird.

multiResVis Steuert die Visualisierung der Multiresolution Struktur.

- **none** M^n wird direkt dargestellt. Die vorhandene Multiresolution-Struktur wird nicht berücksichtigt.
- **nice** Benutzt das progressive Netz direkt zur Darstellung und erzeugt für jeden *vertex-split* einen neuen Punkt in der Punktliste. Generiert das beste Ergebnis aber verändert die Koordinaten bei jeder Veränderung der Auflösung. Da die Koordinaten meist direkt auf der Graphik-Hardware abgelegt sind, kann dies zu Performance-Problemen führen.
- **fast** Erzeugt initial eine vereinfachte *collaps-map* [88] aus dem progressiven Netz. Die *collaps-map* besitzt für jede *edge-collaps* nur einen einzelnen Index der auf die neue Punktposition für v_s verweist. Es wird keine neue Position generiert, sondern immer eine bestehende Koordinaten benutzt. Dieses Verfahren erzeugt weniger korrekte Ergebnisse als *nice,* ist aber wesentlich schneller.
- **auto** Der Knoten liest den multiRedGen Wert nicht aus dem Feld, sondern aus dem aktuellen Kontext. Somit ist es möglich einen Standardwert im Kontext zu setzen, der von allen Knoten mit *auto* Feldwert während der Initialisierung übernommen wird.

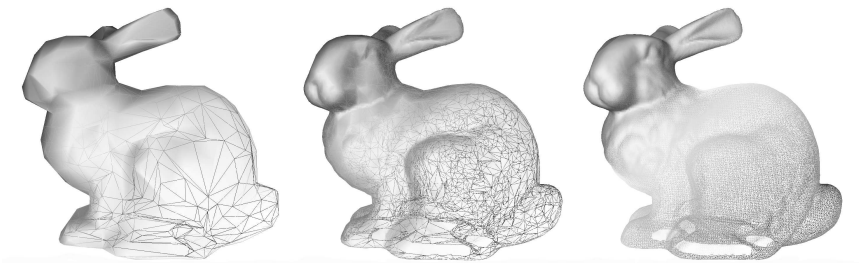

Abbildung 11: Progressives Netz mit 1000, 30000, 65451 Dreiecken

Alternativ zu der IFS-Erweiterung könnte man auch einen neuen Knotentyp speziell zur Kodierung von progressiven Netzen erzeugen. Einen vergleichbaren Ansatz hat Martin Isenburg in seinen Arbeiten zu Netzkompression für X3D vorgestellt, um Netztopologien in ASCII zu kodieren [62][63]. Der Knoten könnte direkt M^0 und nachfolgend alle vertex-split Operationen kodieren und hätten somit den Vorteil, dass er auch für das progressive Übertragen und Verarbeiten geeignet wäre. Jedoch ist es dann nicht mehr möglich, Morpher und Interpolatoren auf den Punkt-Attributen anzuwenden.

.3.1.4 Skalierbarkeit und Ergebnisse

Die meisten Netze in typischen VR-Anwendungen sind *manifold* und ohne Löcher. Somit verringert jede *edge-collapse* Operation die Anzahl der Dreiecke um zwei, jede *vertex-split* Operation erhöht die Anzahl um zwei Dreiecke. Die Darstellungskosten skalieren, wenn man die Füll-Kosten nicht berücksichtigt, linear von M^0 bis M^n. In den meisten *non-manifold* Netzen und/oder Netzen mit Löchern sind die Störungen so gering, dass man im Mittel auch von einer linearen Skalierung ausgehen kann.

Das M^n Model des Stanford-Hasen hat 69451 Dreiecke (siehe Abbildung 11). Wenn man die einzelnen Dreiecke direkt zur Darstellung benutzt ohne die *vertex-cache* Hardware explizit zu nutzten, skaliert die Darstellungszeit nahezu linear (siehe Abbildung 12). Benutzt man eine einzige auf dem M^ngenerierte Liste von *Triangle-Strips*, dann variieren die Darstellungszeiten zwischen 0 und 0.125 Sekunden. Man erreicht somit eine wesendliche Verbesserung vor allem bei hohen Auflösungen. Benutzt man mehrere Auflösungsstufen (in diesen Fall drei), bekommt man die besten Darstellungszeiten, jedoch skalieren die Kosten nicht mehr linear.

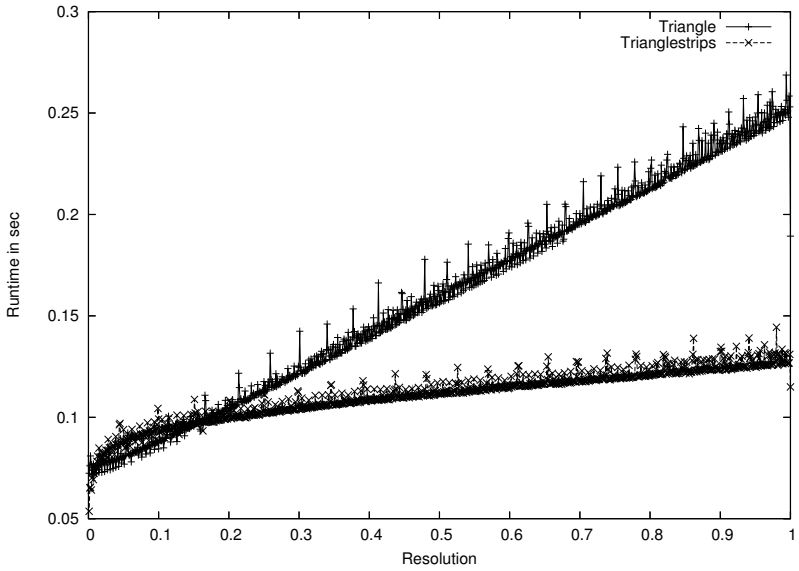

Abbildung 12: Darstellungszeiten für ein Progressives Netz mit Dreiecken und Triangle-Strip Primitiven

Die Messungen wurden auf einem 2 MHz Rechner mit Geforce 4 GPU, standard GL shading und einer Lichtquelle durchgeführt.

.3.2 Volumenvisualisierung

Techniken zur Visualisierung von Volumendaten haben sich in den letzten Jahren zu einem wertvollen Werkzeug für unterschiedliche Applikationen entwickelt [67].

Speziell für dreidimensionale Skalarfelder kristallisiert sich der Bereich der Volumenvisualisierung als geeignete Methode heraus, die ansonsten visuell schwer zugänglichen Strukturen aus Bereichen der medizinischen Diagnose (z.B. CT, MRI, PET, Ultraschall), geophysikalischen Analyse oder numerischen Simulation darzustellen.

.3.2.1 Grundlagen

Der Begriff Volumenvisualisierung [67] bezeichnet die Repräsentation, Manipulation und Darstellung von Volumendaten. Er umfasst alle Stufen der Visualisierungspipeline angefangen vom *Filtering* und *Mapping*, bis zur algorithmischen Bildgenerierung, die auch als *Rendering* bezeichnet wird. Im Zusammenhang mit der Erzeugung von Bildern aus Volumendaten wird deshalb häufig von Volume-

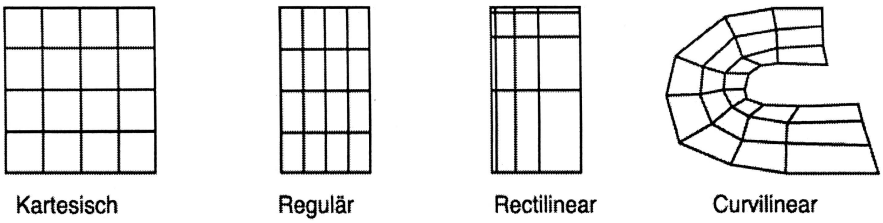

Abbildung 13: Strukturierte Volumengitter

Visualization und Volume-Rendering gesprochen.

.3.2.1.1 Volumendaten 3D-Skalarfelder als gebräuchlichste Form von Volumendaten enthalten zu jedem Punkt im Raum $\mathbb{R}^3$ genau einen skalaren Wert s.

$$s = f(x, y, z);\ x, y, z \in \mathbb{R} \tag{5}$$

Die räumliche Zuordnung der durch s beschriebenen Objekteigenschaften ist durch die x, y und z-Koordinaten eindeutig bestimmt und als Elemente eines 3D-Gitters abgelegt. Jede Zelle dieses Gitters besitzt eine räumliche Ausdehnung und wird *Voxel* genannt.

Da Volumendaten aus Messungen und Simulationen nur an diskreten Gitterpositionen im Raum gegeben sind, werden zur Bestimmung skalarer Werte an beliebigen Stellen in Raum benachbarte Datenpunkte interpoliert. Volumendaten werden in einer Reihe wissenschaftlicher Anwendungen in unterschiedlicher Weise ermittelt: In Simulationen werden aufgrund eines mathematischen Modells physikalische Vorgänge simuliert und damit beispielsweise die Dichtewerte eines Gases an verschiedenen Raumpunkten während einer Strömungssimulation ermittelt. Messungen bestimmen physikalische Größen im Raum mit entsprechender Sensorik, wie zum Beispiel die Abschwächung der Röntgenstrahlung durch einen Computertomographen in der medizinischen Bildverarbeitung. Oft lassen sich Volumendaten auch direkt berechnen, so im Fall der Aufenthaltswahrscheinlichkeit von Elektronen um einen Atomkern oder in einem Molekül, die zur Visualisierung von Molekülorbitalen benötigt wird.

Grundsätzlich lassen sich die Datengitter in strukturiert und unstrukturierte Gittertypen aufteilen.

Strukturierte Gittertypen (siehe Abbildung 13) besitzen eine gleichmäßige

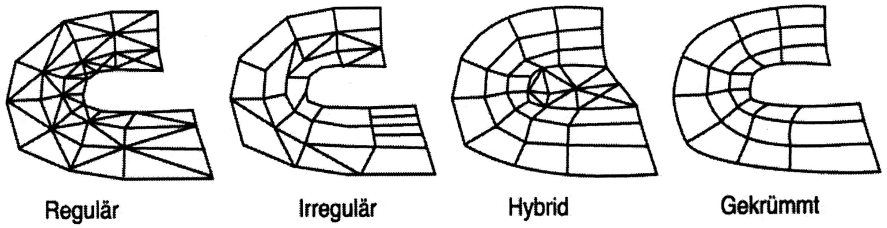

Abbildung 14: Unstrukturierte Volumengitter

Nachbarschaftsstruktur. Bei kartesischen bzw. regulären Gittern liegen Zellen als regelmäßige Quader in einheitlicher Größe vor. Bei rectilinearen Gittertypen handelt es sich immer um quaderförmige Zellen, jedoch können diese unterschiedliche Größen annehmen. Die Zellen in einem curvilinearen (krummlinigen) Gitter sind nicht mehr quaderförmig, sondern verformte Hexaeder.

Unstrukturierte Gitter (siehe Abbildung 14) besitzen keine besondere Anordnungsstruktur, deshalb ist ein direkter Zugriff auf einzelnen Zellen nicht möglich. Der Datenraum besteht aus einer Menge von räumlich positionierten Knoten und einer Menge von Zellen, die diese Knoten miteinander verbinden. Gewöhnlich werden Gitter mit einer einheitlichen Zell-Topologie verwendet, wie z.B. Tetraedergitter oder Hexaedergitter. Es können aber auch verschiedene Zellformen in einem Gitter kombiniert werden.

Die weitaus gebräuchlichste Form der Volumendaten sind strukturierte Gitter in kartesischer Form. Die meisten Messverfahren und Techniken (zum Beispiel CT oder MR) erzeugen Daten in dieser Form. Auch ist es möglich, alle weiteren Gitterformen durch *resampling* in kartesische zu transformieren.

Aus diesem Grund beschränken sich fast alle Verfahren zur Echtzeitvisualisierung von Volumendaten auf die Verarbeitung von kartesischen Gittern.

.3.2.1.2 Verfahren zur Volumenvisualisierung Seit ihrer Einführung in den späten 1980er Jahren wurden im Umfeld der Volumenvisualisierung einige mehr oder weniger unterschiedliche Techniken entwickelt, die sich grob in vier Klassen einteilen lassen [137]:

Object-order Algorithmen Diese Methoden, wie zum Beispiel Marching Cubes [81] oder Marching Tetraeder, suchen eine polygonale Umsetzung des Skalarfeldes, um eine Flächenstruktur zu erzeugen. Die durch Abtasten des Vo-

lumens entstandenen Polygonnetze sind beliebig feine Approximationen des Originals. Methoden dieser Gruppe werden auch als *indirektes Volumerendering* bezeichnet, da sie zur Darstellung nicht mehr direkt die Originaldaten verarbeiten sondern die Polygonnetze. Werden die Volumendaten unmittelbar verarbeitet, so wie bei den nachfolgenden Verfahren, spricht man vom *direktem Volumerendering.*

Image-order Algorithmen Ausgehend von den Bildpunkten des Zielbildes werden Strahlen in die Szene geschickt, die den Beitrag des Volumens zu einem Punkt der Bildebene ermitteln. Werden nur primäre Sichtstrahlen betrachtet so wird vom Ray-Casting-Verfahren [67] gesprochen. Da der Strahl mindestens solange verfolgt wird, bis sich der erreichte Farbwert nicht mehr weiter ändert, ist diese Methode sehr rechenintensiv und langwierig. Allerdings wird so die bisher beste Bildqualität erreicht. Je nach Detaillierungsgrad können auch physikalische Eigenschaften wie zum Beispiel Refraktion und Spiegelung berücksichtigt werden.

Shear-Warp Factorization Durch Faktorisierung der Blick-Transformation in zwei Einzeltransformation (eine Scherung und eine Verzerrung) kann das Bild mit Hilfe von Volumenscheiben berechnet werden [76][64]. Shear-Warp kombiniert image-order und object-order Ansätze und bietet die Möglichkeiten der Parallelisierung und Scanline-Kompression.

Texture-basierte Algorithmen Hierbei wird die Texturfähigkeit der Graphikhardware (*graphics processing unit; GPU*) verwendet, um die Volumendaten selbst oder berechnete Farbwerte zu speichern. Die Darstellung geschieht durch das Überlagern und Überblenden von Polygonflächen, die texturierte Scheiben des Volumens repräsentieren.

Indirect Volumen Rendering Techniken sind nur bedingt als allgemeines Verfahren geeignet, da sie nicht unmittelbar die 3D-Skalardaten darstellen, sondern nur ausgewählte ISO-Flächen. Die direkten Volumenrendering-Verfahren unterscheiden sich vor allem in Laufzeitverhalten und Qualität der erzeugten Bilder. Softwareverfahren erzeugen im allgemeinen bessere Ergebnisse auf Kosten der Laufzeit. Hardware-unterstützte Verfahren wie die Texture-basierten Algorithmen, sind im allgemeinen Performanter.

Bis vor wenigen Jahren hat die Hardware-Shader-Pipline die Möglichkeiten der Umsetzung von Volumen-Rendering Verfahren weitgehend eingeschränkt. Durch

die Einführung der Vertex und Fragment-Shader [68] ist es möglich, alle prä-/post-Klassifikationen und Integrationen direkt als Shaderprogramm umzusetzen. Somit sind heute Texture-basierte Verfahren bevorzugt, wenn es darum geht qualitative hochwertige Volumen in Echtzeit darzustellen. Auch hier wird dieses Verfahren für die Implementierung und weitere Untersuchung ausgewählt.

.3.2.1.3 Physikalische Grundlagen Gemeinsam ist allen Verfahren der direkten Volumenvisualiserung die (approximative) Auswertung des Volume-Rendering-Integrals [72] für jeden Bildpunkt des Zielbildes, also die Integration von abgeschwächten Farb- und Absorptionskoeffizienten entlang eines Sichtstrahls.

Im Folgenden wird davon ausgegangen, dass der Sichtstrahl $x(\lambda)$ in Abhängigkeit vom Abstand zur Betrachterposition parametrisiert ist und dass die Farbdichten $color(x)$ zusammen mit den Absorptionsdichten $extinction(x)$ für jeden Punkt im Raum x gegeben sind. Die Einheiten der Farb- und Absorptionsdichten sind Farbintensität und Absorptionsstärke pro Längeneinheit. Für jeden Sichtstrahl ergibt sich die resultierende Intensität I aus der Volume-Rendering-Gleichung:

$$I = \int_0^D color(x(\lambda)) \exp^{-\int_0^\lambda extinction(x(\lambda))d\lambda} d\lambda \tag{6}$$

mit der maximalen Distanz D, d.h. es existiert keine Farbdichte $color(x(\lambda))$ für λ größer als D und kleiner als 0 (siehe Abbildung 15). Die Gleichung drückt aus, dass die Farbe an jedem Punkt x bezüglich der Funktion $color(x)$ emittiert wird und durch die integrierten Absorptionskoeffizienten $extinction(x)$ zwischen dem Blickpunkt und dem Emissionspunkt abgeschwächt wird.

Leider ist diese Form der Volume-Rendering-Gleichung nicht zur Beschreibung der Visualisierung eines kontinuierlichen Skalarfeldes $s(x)$ geeignet, da die Berechnung der Farb- und Absorptionskoeffizienten nicht spezifiziert ist. Zwei Schritte werden bei der Ermittlung dieser Farb- und Absorptionskoeffizienten unterschieden: Die *Klassifikation* ist die Zuweisung einer primären Farbe und des Absorptionskoeffizienten zu einem Skalarwert. Eine Klassifikation wird durch die Einführung von *Transferfunktionen* für die Farbdichten $\tilde{c}(s)$ und Absorptionsdichten $\tau(s)$ erreicht, welche die Skalarwerte aus dem Volumen $s = s(x)$ auf Farb- und Absorptionskoeffizienten abbilden. Allgemein ist dabei $\tilde{c}$ ein Vektor, der die Farbe in einem beliebigen Farbraum angibt, während es sich bei τ um einen skalaren

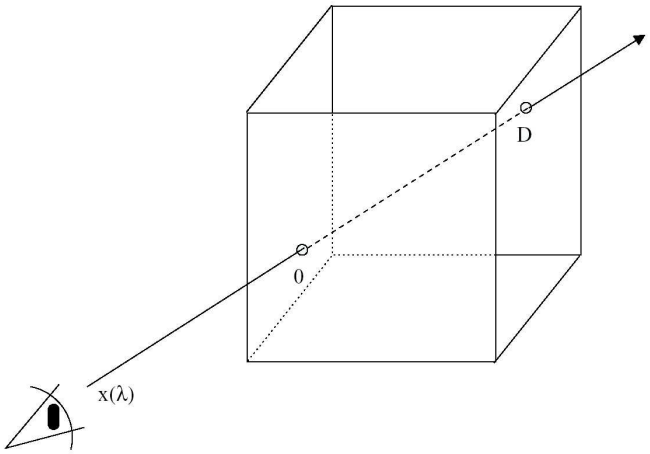

Abbildung 15: Parameter des Volumen-Rendering-Integrals

Absorptionskoeffizienten handelt.

Der zweite Schritt ist das Shading, das den Beitrag eines Punktes im Raum, also die Funktion $color(x)$ berechnet. Das Shading hängt natürlich von der Primärfarbe ab, kann aber auch von anderen Parametern, wie beispielsweise dem Gradienten $s(x)$ des Skalarfelds, ambienten und diffusen Lichtparametern, etc. beeinflusst werden. Im weiteren Verlauf dieses Abschnitts wird allein die Klassifikation eine Rolle spielen. Aus diesem Grunde wird zunächst von einem trivialen Shading ausgegangen, d.h. $color(x)$ wird durch die in der Klassifikation zugewiesene Primärfarbe $\tilde{c}(s(x))$ ersetzt. Analog wird $extinction(x)$ durch den in der Klassifikation zugewiesenen Absorptionskoeffizienten $\tau(s(x))$ ersetzt. Mit diesen Vereinbarungen kann die Volume-Rendering-Gleichung in die folgende Form gebracht werden:

$$I = \int_0^D \tilde{c}(s(x(\lambda))) \exp^{-\int_0^\lambda \tau(s(x(\lambda')))d\lambda'} d\lambda \tag{7}$$

.3.2.1.4 Prä- und Post-Klassifikation Direkte Volumenvisualisierungstechniken unterscheiden sich wesentlich in der Auswertung von Gleichungen 7. Ein wichtiger und wesentlicher Unterschied ist die Berechnung von $\tilde{c}(s(x))$ und $\tau(s(x))$. Tatsächlich ist das kontinuierliche Skalarfeld $s(x)$ üblicherweise durch ein Gitter mit skalaren Werten s_i an jedem Gitterpunkt v_i beschrieben, aus dem mit einer Interpolationsvorschrift alle weiteren Punkte berechnet werden können.

Die Reihenfolge der Interpolation und der Anwendung der Transferfunk-

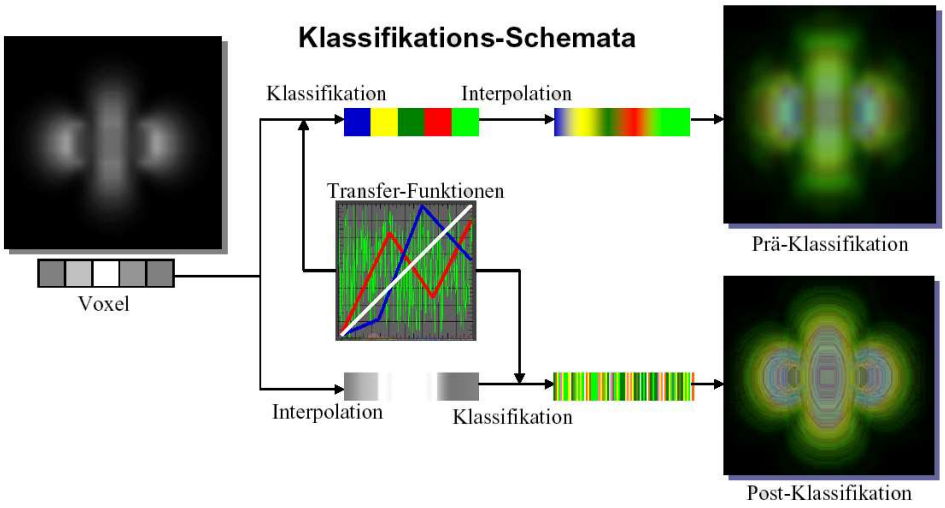

Abbildung 16: Prä- und Post-Klassifikation

tionen definiert den Unterschied zwischen Prä- und Post-Klassifikation. Post-Klassifikation ist durch die Anwendung der Transferfunktionen *nach* der Interpolation von $s(x)$ aus den umgebenden Skalarwerten charakterisiert. Im Gegensatz dazu wird bei der Prä-Klassifikation die Transferfunktion *vor* dem Interpolationsschritt angewandt, d.h. Farben $\tilde{c}(s_i)$ und Absorptionskoeffizienten $\tau(s_i)$ werden in einem ersten Schritt für jeden Gitterpunkt v_i nachgeschlagen und dann für die Berechnung des Volume-Rendering-Integrals $\tilde{c}(s(x))$ und $\tau(s(x))$ interpoliert.

Offensichtlich produzieren Prä- und Post-Klassifikation unterschiedliche Ergebnisse, sobald die Interpolation zweier Skalarwerte des Volumens nicht mit der Anwendung der Transferfunktionen vertauschbar ist. Da die Interpolation üblicherweise nichtlinear ist (beispielsweise tri-linear in kartesischen Gittern), kann diese nur dann mit der Transferfunktionen vertauscht werden, wenn die Transferfunktion eine lineare Funktion durch den Ursprung oder die Identitätsfunktion ist. In allen anderen Fällen führt Prä-Klassifikation zu Abweichungen von der Post-Klassifikation (siehe Abbildung 16). Letztere kann in dem Sinne als korrekt angesehen werden, als sie die Anwendung einer Transferfunktion auf ein kontinuierliches Skalarfeld annimmt, das durch ein Gitter zusammen mit einer Interpolationsvorschrift gegeben ist. Trotzdem ist Prä-Klassifikation unter bestimmten Umständen nützlich, da sie als eine grundlegende Segmentierungstechnik eingesetzt werden kann.

.3.2.2 Texturbasierte Verfahren

Die Idee der Verwendung von 3D-Texturen wurde in der Literatur erstmals von Kurt Akeley erwähnt [2]. Timothy Cullip und Ulrich Neumann stellten eine Methode zu Rekonstruktion von CT-Scan-Data für die RealityEngine vor und setzten somit die Idee um[32].

Die Darstellung von beleuchteten Volumen stellt eine besondere Herausforderung dar. Da bisher keine Möglichkeit bestand, weiteren Einfluss auf die hardwareinterpolierten Daten zu nehmen, waren nur softwaretechnische oder hardwaretechnische On-Chip Lösungen verfügbar[115]. Allen Van Gelder und Kwansik Kim untersuchten in ihrer Arbeit [137] unterschiedliche Architekturen für Renderingmethoden mit Beleuchtung und definierten einen optimalen Aufbau. Jedoch war es Mitte der 90er nicht möglich, diese optimale Architektur mit Standardhardware umzusetzen.

Rüdiger Westermann und Thomas Ertl beschleunigten [146] die bisherigen Methoden durch intensiven Einsatz von Fragment-Operationen wie zum Beispiel *Stencil-Tests* und *Alpha-Test*s. Beleuchtung wurde durch Definition einer Farb-Matrix erreicht. Allerdings setzten die meisten dieser Ansätze Multi-Pass Rendering voraus, d.h. das graphische Ergebnis ist erst nach mehreren Durchläufen vollständig.

Dachille et al. realisierten eine neue Rendering Architektur [33]. Die Interpolationseigenschaft der Graphikhardware wurde hierbei aufgrund ihrer Leistungsfähigkeit eingesetzt, das Ergebnis wurde dann aber wieder in den Hauptspeicher übertragen. Das endgültige Bild entstand schließlich mit Hilfe der *Sweep-Planes* Methode, mit der jede Scanlinie aus einer texturierten Fläche berechnet wird, die senkrecht zur Bildebene steht.

Zur Darstellung von großen Datensätzen, deren Ausmaße über die Texturgrösse hinausgeht, schlugen LaMar et al. eine Multiresolution-Technik vor, welche interaktive Darstellungsarten garantieren [76].

Basierend auf der *Projected Tetrahedra Algorithmus (PT)* von Shirley und Tuchmann [122] entwickelten Röttger et al. eine Methode für Tetraeder-Volumen-Zellen unter Verwendung von integrierten 2D- und 3D-Texturen [110].

Die Einschränkungen der oben bsiher Verfahren lagen unter anderem darin, dass nach einer Flächentexturierung keine weitere Operation im Rendering-Prozess durchgeführt werden konnte. Dies änderte sich durch die Einführung programmierbarer Graphik-Pipline-Stufen, sogenannte *Shader*. Dadurch wurde es möglich, mit mehreren Texturen gleichzeitig vollwertige, mathematische Berech-

nungen während der Rasterung pro Pixel durchzuführen. Die Industrie entwickelte zu Beginn unterschiedliche Realisierungsvarianten für Shader. Die Anwendungen dieser Shader für Volumenrendering wurde von Resk-Salama et al. [109] untersucht .

Klaus Engel et al. kombinierten 2001 die Methoden von Van Gelder und Kim, Röttger et al. und Resk-Salama et al. zu einem hochqualitativen, hardwarebeschleunigten Renderingverfahren[38]. Die vollständige Beleuchtungsberechnung wird dabei in Echtzeit auf der Hardware durchgeführt.

In [14] wurde eine effiziente Methode zur Visualisierung von dynamischen Volumen vorgestellt.

Allen Verfahren gemeinsam ist die Umsetzung des Rendering-Integrals mit Hilfe von Schnittpolygonen, die einzelne Schichten durch das Volumen abbilden. Diese Schichten, auch *Slices* genannt, sind entweder Object-aligned und statisch für 2D-Texturen oder Viewport-aligned und somit dynamisch für 3D-Texturen.

Object-aligned Slicing Stehen nur 2D-Texturen zur Verfügung, muss das Verfahren *object-aligned slices* erzeugen, d.h. die Schnittpolygone werden als Stapel von 2D-Texturen einmalig vorberechnet. Es werden drei Schicht-Stapel verwendet, deren Schichten jeweils parallel zur $xy-$, $xz-$ und $yz-$ Ebene liegen. Dabei wird der Schichtstapel für die Darstellung ausgewählt, bei dem der Winkel zwischen der Schicht-Normalen und dem Kameravektor minimal ist. Die Schichten eines Stapels werden aus Sicht des Betrachters von hinten nach vorne unter Verwendung der Hardwareoperationen im Bildspeicher überblendet, wobei nur bilineare Interpolation auf den einzelnen Schichten zur Anwendung kommt (siehe Abbildung 17).

Vorteile dieses Verfahrens sind neben der besseren Verfügbarkeit von 2D-Texturen die im Allgemeinen höherer Bildwiederholrate. Der jedoch entscheidende Nachteil ist, dass die Abtastrate nicht anpassbar variiert und nur bilineare Interpolation unterstützt.

Viewport-aligned Slicing Die Verwendung von 3D-Texturen ermöglicht es, einzelne Pixelwerte der Schnittpolygone tri-linerar zu interpolieren. Dabei werden die Schnitte dynamisch, abhängig von der Kameraposition und Orientierung errechnet (siehe Abbildung 18). Diese *viewport-alienged slices* clippen das Volumenrechteck und liefern 3- bis 6-seitige Polygone. Den einzelnen Eckpunkten der Polygone werden 3D-Texturkoordinaten zugewiesen. Somit werden die Pixel der Schnitt-

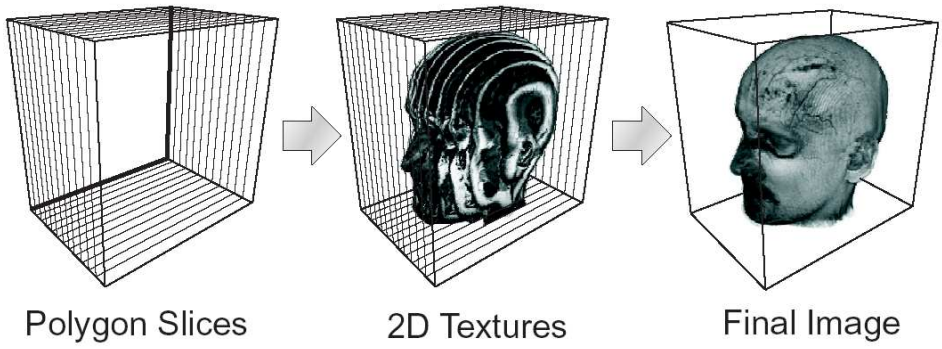

Abbildung 17: View-alligned Schnitte für 2D-Texturen

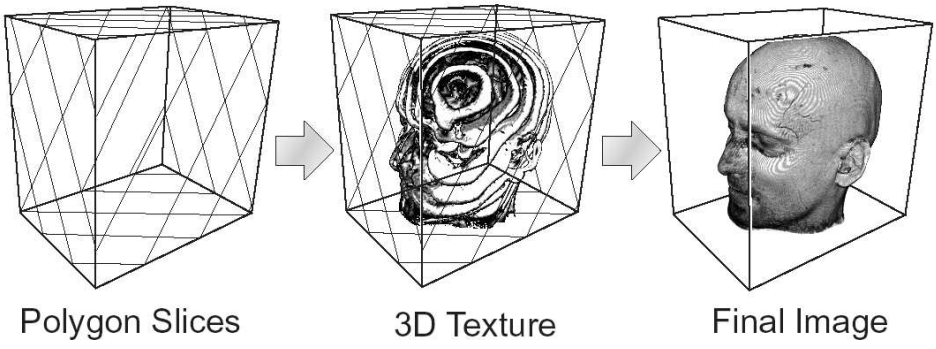

Abbildung 18: Viewport alligned Polygone für 3D-Texturen

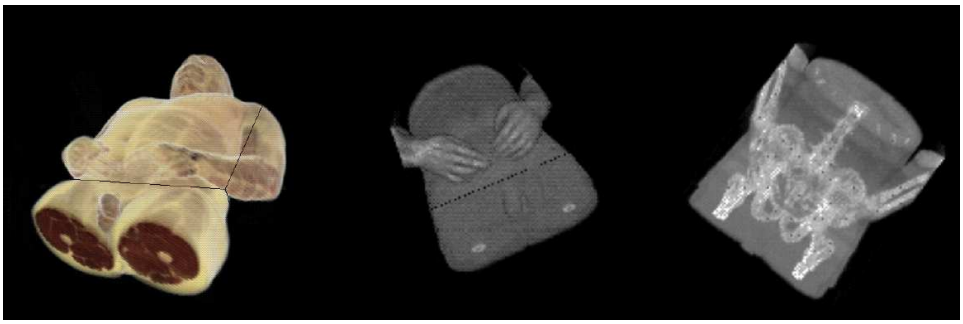

Abbildung 19: Anwendung von unterschiedlichen Blend-Funktionen zur Volumenvisualisierung

flächen für jedes Bild und jede beliebige Position und Orientierung direkt aus dem Volumenblock ermittelt. Durch die Anwendung von geeigneteren Alpha und Blend Funktionen der Hardware entsteht das endgültige Bild aus den einzelnen Schnitten (siehe Abbildung 19).

Vorteil diesen Verfahrens ist die Unterstützung von trilinearen Interpolationen bei konstanter Abtastrate. Nachteil ist die Voraussetzung von 3D-Texturen. Dieses Feature war bis vor kurzen noch nicht auf allen Hardwareplattformen verfügbar.

.3.2.3 Integration

In diesem Abschnitt werden auf der Grundlage der X3D-Spezifikation [143] Erweiterungen entwickelt mit dem Ziel, direktes Volumen-Rendering in das Rahmensystem zu integrieren. Diese Strukturen erweitern die Ergebnisse aus [10][14] und erlauben somit eine direkte Nutzung von aktueller Shader-Hardware.

Wie in Abschnitt 5.4.2.1.1 festgestellt, sind Volumen-Daten in den meisten Fällen als reguläre Gitter abgelegt. Der einzige Datentyp in der X3D-Spezifikation, der regulare Gitter direkt unterstützt, ist der 2D-Image-Feldtyp. Es scheint nur natürlich, die 2D-Image Definition so zu erweitern, dass sie beliebige Dimensionen unterstützt. Darüberhinhaus ist es notwendig einen neuen Knoten einzuführen, der die texturierten Polygonenschichten generiert.

.3.2.3.1 1D/2D/3D Textur-Knoten Texturdaten sind in X3D als 2D-Koordinatensystem $(s,t) \in [1,2]^2$definiert. In der Spezifikation existieren drei unterschiedliche Knotentypen für Texturdaten. Der *ImageTexture* Knoten lädt genau ein Bild mit 2D-Pixelwerten von einer externen Quelle oder Datenstrom.

Der *MovieTexture* Knoten definiert animierte Bilddaten und lädt 2D-Pixelwerte ebenfalls von einer externen Quelle. Der *PixelTexture* Knoten erlaubt dem Benutzer Pixelwerte direkt als Feldwerte anzugeben. Nur der *PixelTexture* Knoten definiert implizit zwei Dimensionen durch das Parametermuster. Die *ImageTexture*- und *MovieTexture*-Knoten können durch eine simple Ergänzung, ein *reapeatR* Feld, auf 1D-, 2D- und 3D-Daten erweitert werden.

```
ImageTexture : StillTexture {
...
  SFBool     []       repeatR TRUE
...
}
MovieTexture : DynamicTexture {
...
  SFBool     []       repeatR TRUE
...
}
```

Analog zu den 2D-Texturen muss es möglich sein, Daten mit bis zu vier Kanälen zu speichern. Einfache Grau-Bilder haben genau einen Kanal. Bilder mit zwei Kanälen definieren einen Grau- und einen Alpha-Wert pro Pixel. Farb-Bilder mit drei Kanälen beschreiben einen RGB-Tripel pro Pixel. Farb-Bilder mit vier Kanälen haben zusätzlich zu den RGB-Anteilen noch einen Alpha-Wert.

Im Gegensatz zu den 2D-Bildformaten existiert zur Zeit kein anwendungs- und systemunabhängiges Dateiformat für 3D-Bilddaten. Für TIFF [29] existiert eine 3D-Erweiterung, die sich jedoch nicht etablieren konnte. Das DICOM [4] hat eine große Bedeutung im Umfeld von medizinischen Anwendungen erlangt, aber nicht darüber hinaus.

Das DDS-Format [90] ist ein spezielles Bildformat für Texturen und unterstützt sowohl 1D-, 2D- und 3D-Bilder. Es hat sich aber bisher nur im Umfeld von Spielen etabliert.

Die Implementierung des Rahmensystems unterstützt zur Zeit 3DTIFF- und DDS- Bilddaten direkt, ist aber offen für weitere Formate die sich in Zukunft etablieren.

.3.2.3.2 SliceSet Knoten Die *Shap*e-Knoten sind die Bausteine für sichtbare Elemente der X3D-Szenenbeschreibung. Sie halten selber keine Objektdefinitio-

nen, sondern verweisen auf *Appearance*- und *Geometry*-Instanzen. Die *Appearance*-Knoten halten wiederum Instanzen von *Material*- und vor allem *Texture*-Knoten.

Die *SliceSet*-Erweiterung die in dieser Arbeit vorgestellt wird, ist eine Spezialisierung des abstrakten *Geometry* Typs. Der Knoten generiert, abhängig von aktueller Position, Orientierung und Parametern, dynamisch Schichtpolygone mit angepassten Texturkoordinaten.

```
SliceSet : Geometry {
...
  SFFloat    [in,out]       resolution   1.0
  SFVec3f    [in,out]       size         1 1 1
...
}
```

Eine *SliceSet* Instanz kann nur als Kind eines *Shape* Knotens angelegt werden. Es ist aber erlaubt, eine *SliceSet* Instanz an beliebig vielen Stellen als *Shape*-Kind wiederzuverwenden.

Das *SliceSet* hat neben den allgemeinen Geometrieparametern zwei weiterer Felder:

size Bestimmt die Dimensionen des Objekts im lokalen Koordinatensystem. Die Werte skalieren das Gitter (s,t,r) und definieren somit die Größe relativ zu der umschließenden Szene.

resolution Definiert die objektabhängige Skalierung der Auflösung R_n (siehe Formel 1).

Der SliceSet Knoten erzeugt die einzelnen Polygonschnitte und rendert und sortiert diese. Er stellt keine Mechanismen bereit, die die Klassifikation oder Beleuchtung direkt steuern. Es ist möglich, den Knoten mit der Standard-Hardware-Pipeline einzusetzen, um Prä-Klassifizierte Volumen unbeleuchtet zu rendern:

```
Shape {
  appearance Appearance {
    texture ImageTexture {
      url ''volume.dds''
    }
```

```
  geometry SliceSet { }
}
```

Verfahren zur Post-Klassifikation und Beleuchtung der Volumendaten sind als Kombinationen von Vertex- und Fragment-Shadern [68] implementiert. Diese Shader-Programme werden als Prototypen bereitgestellt und sind somit auf der einen Seite sehr effizient und auf der anderen Seite auch sehr leicht erweiterbar, ohne die Frameworkknoten verändern zu müssen.

Zur Zeit existieren unterschiedliche Prototypen, um die Shader-Implementierung zur Prä- und Post-klassifikation, Beleuchtung mit vorberechneten Gradienten und Beleuchtung mit Gradientenbestimmung zur Laufzeit zu unterstützen.

Die Prototypen sind spezielle *Appearance*-Knoten und als solche direkt als Kind einer Shape-Instance zusammen mit einem SliceSet einsetzbar:

```
Shape {
  appearance PostLitVolApp {
    volumeURL       ''volume.dds''
    transferFuncURL ''trans.png''
  }
  geometry SliceSet {}
}
```

In diesem Beispiel wird der Gradient, als Grundlage für die Beleuchtungsberechung, zur Laufzeit bestimmt. Die Shader selbst sind in GLSL [68] implementiert, was spezielle Anforderungen an die Hardware stellt.

.3.2.4 Skalierbarkeit und Ergebnisse

Texturebasierte Volumenrendering-Verfahren (siehe Abschnitt .3.2.2) sind immer auch Multiresolution-Methoden, wenn die Abtastrate zur Laufzeit frei gewählt werden kann. Das ist in fast allen Fällen so, ausgenommen in Verfahren, die nicht mit einzelnen Schichten sondern Prä-Integrierten Steifen arbeiten [38]. In diesen Fällen müssen die Prä-Integrierten Lookup-Tabellen für jede Schichtdichte neu berechnet werden. Alternative kann man 3D-Lookup-Tabellen (Die Schicktdicke als eigene Dimension kodieren) oder 1D-Approximationen mit entsprechend aufwendigen Fragment-Shadern benutzen.

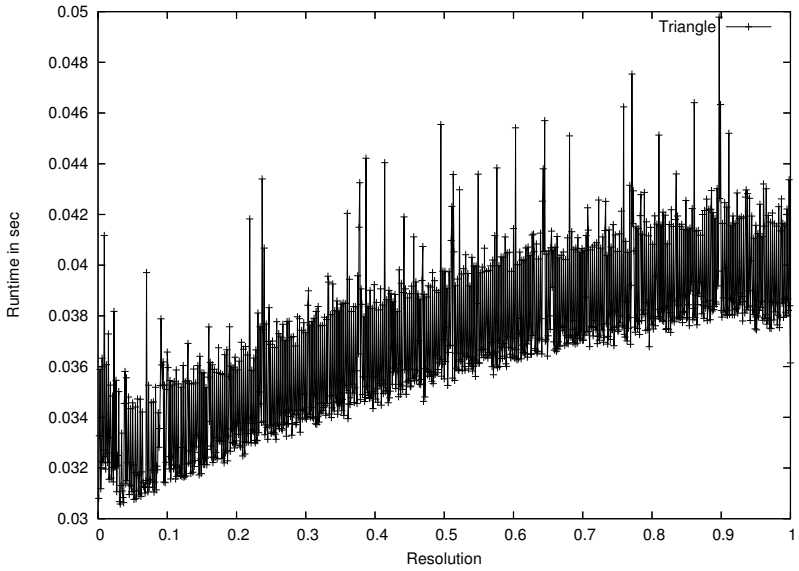

Abbildung 20: Lineare Abhängigkeit der Darstellungszeiten für Volumen

Im Idealfall sollte die Abtastrate der Auflösung des ursprünglichen Volumens entsprechen. Aus diesem Grund wird für eine resultierende Auflösung R_r (siehe Formel 1) von 1 eine Schichtanzahl gewählt, der Gitterauflösung g entsprechend. Ansonsten wird die Gitterauflösung mit der resultierenden Auflösung skaliert ($Schicht\ Anzahl = g * r$).

Entspricht der Schichtabstand nicht der ursprünglichen Abtastrate so ist es notwendig, auch den Absorptionsgrad zu skalieren [144][145]. Da der Absorptionsgrad aber erst im Fragment-Shader bestimmt wird, ist diese Skalierung nicht vom allgemeinen *SliceSet* Knoten bestimmbar. Aus diesem Grund übergibt der *SliceSet* Knoten die aktuelle Schichtdicke und die aktuelle resultierende Auflösung an den Shader. Der Fragment-Shader muss dann selbständig den Absorptionsgrad pro Voxel skalieren.

Texturbasiertes Volumenrendering ist durch die Verwendung von 3D-Texturen und der aufwendigen Interpolationen auf den Volumendaten fast immer *fill-limited*. Das heißt, dass die Berechnung und Transformation der Eckpunkte der einzelnen Schichten nicht ins Gewicht fällt. Ansonsten skaliert die Performance des Verfahrens nahezu linear mit der Anzahl der Schichten beziehungsweise der Auflösung R_r (siehe Abbildung 20). Die Abweichungen und Schwankungen ergeben sich durch das nur schlecht vorhersehbare Speicher- und Cache-Verhalten der Graphikhardware.

Die Messungen wurden auf einen 2 MHz Rechner mit Geforce 4 GPU und einem 256^3Volumen ohne Beleuchtungsrechnung durchgeführt.

.4 Zusammenfassung

Anhand von aktuellen Entwicklungen in der Prozessortechnologie wird gezeigt, dass es für die optimale Auslastung der Ressourcen notwendig ist, möglichst viele Aufgaben und Verfahren zur Laufzeit auf unterschiedliche Prozesse aufzuteilen. Dazu werden die typischen Abläufe in einem VR-System untersucht und eine Lösung entwickelt, die zwei unterschiedliche Ansätze der Parallelisierung beinhaltet. Zum Einen wird die klassische Trennung der Applikation und Darstellung unterstützt und verallgemeinert, und zum Anderen ein neuartiges Verfahren entwickelt und vorgestellt, das die automatische Parallelisierung auf den Ereigniskanten der Applikationsgraphen bereitstellt. Der entscheidende Vorteil gegenüber den bestehenden Verfahren ist die selbständige und autarke Ermittlung von Teilgraphen, die parallel abgearbeitet werden können.

Mithilfe dieser Lösung können Applikationen sehr effizient verteilt werden. Dennoch sind die Ressourcen realer Maschinen begrenzt. Um globale Applikationsvorgaben erfüllen zu können, wie zum Beispiel das Darstellen mit einer fixen Bildwiederholrate oder die Animation einer variierenden Anzahl von Figuren pro Zyklus, ist eine globale Skalierbarkeit der Kosten gefordert. Zur Lösung dieses Problems wird ein globales Modell entwickelt und eingeführt, das automatisch lokale Kosten der Prozesse zur Darstellung und Veränderung skaliert. Um die Voraussetzung erfüllen zu können, werden unterschiedliche skalierbare Verfahren untersucht und teilweise entscheidend verbessert.

Das in dieser Arbeit entwickelte Modell zur Parallelisierung und Skalierung von MR-Systemen ist eine wesentliche Verbesserung gegenüber den bisherigen Verfahren, da es auf der Grundlage der globalen Zielvorgaben und dem Applikationsgraphen selbständig und automatisch den Ressourcenverbrauch regelt und verteilt. Somit wird der Prozess der Anwendungsentwicklung wesentlich vereinfacht, da eine aufwendige und fehlerträchtige manuelle Parallelisierung entfällt.

Anhang A

Applikationsbeispiele

Das in dieser Arbeit vorgestellte Rahmensystem und dessen Implementierung für Mixed-Reality-Anwendungen stellt die Basis für eine Vielzahl von Projekten dar. Beispielhaft werden im Folgenden drei von ihnen vorgestellt: zwei Virtual-Reality-Anwendungen (“Dom von Siena” und “Embassi-Fahrsimulator”), sowie eine Augmented-Reality-Anwendung (“Archeoguide”).

Weitere Anwendungen und Informationen über den aktuellen Entwicklungsstand des Systems sind auf der Projektseite im Internet [60] zu finden.

A.1 Dom von Siena

Der Dom von Siena [13] (siehe Abbildung A.1) ist eine klassische VR-Anwendung im Infotainment- und Cultural-Heritage-Bereich. Bekannt ist, dass der moderne Massentourismus bei vielen historischen Sehenswürdigkeiten an seine Grenzen stößt. Häufig kann nur eine begrenzte Anzahl von Personen gleichzeitig eine Sehenswürdigkeit besuchen oder der öffentliche Zugang zu historischen Stätten muss ganz gesperrt werden, da die von den Menschenströmen verursachten Schäden einfach zu groß sind. Gleichzeitig stellt der Tourismus einen wichtigen Wirtschaftsfaktor für die häufig unterentwickelten Gebiete dar, in denen sich die historischen Stätten befinden.

Der Konflikt zwischen den Bestrebungen, kulturelle Schätze zu vermarkten, und den Bestrebungen, sie für die Zukunft zu konservieren, kann durch den Einsatz von modernen VR- und AR-Technologien gelöst werden. Anstatt Besucher durch die realen Stätten zu führen, werden virtuelle Nachbildungen verwendet.

So konnten die Besucher der Weltausstellung EXPO 2000 sich auf den vir-

 L

Abbildung A.1: Anwendung der Gesichts- und Körperanimationstechniken

tuellen Rundgang durch ein Modell des Domes von Siena begeben. Das Modell besteht aus 250.000 Polygonen und 250 MB Texturen, die Anwendung wurde als Stereoprojektion realisiert.

Technisches Highlight ist der virtuelle Reiseführer Luigi, der die Besucher durch den Dom führt und ihnen Informationen aus den Bereichen Kunst, Geschichte und Architektur präsentiert. Für diesen sogenannten *Avatar* wurden modernste Techniken aus der Computergraphik wie *Morphing* und *Skin-and-Bone*-Systeme eingesetzt.

In herkömmlichen VR-Anwendungen müssen alle Animationen von Hand erzeugt werden, d.h. jede Bewegung des Avatars, jedes Minenspiel seines Gesichts und jeder Faltenwurf seines Gewands muss manuell von einem Modellierer erzeugt werden. Beim Reiseführer Luigi ist das nicht der Fall. Das Rahmensystem ermöglicht, dass große Teile der Animation in Echtzeit während der Präsentation vom Computer berechnet werden.

Abbildung A.2: Archeoguide; eine Freiland-AR Anwendung

A.2 Archeoguide

Ein weiteres Anwendungsbeispiel aus dem Bereich Cultural Heritage ist das Projekt Archeoguide (siehe Abbildung A.2). Archeoguide lässt mit den Mitteln modernster AR-Technik die Tempel und Sportstätten von Athen auf dem Gelände des alten Olympia auferstehen.

Der Besucher von Olympia findet heute nur noch Grundmauern der alten Gebäude vor. Der Lauf der Zeit hat alle Bauwerke buchstäblich dem Erdboden gleich gemacht. Es erfordert eine Menge Phantasie sich vorzustellen, wie hier vor langer Zeit Tempel standen, Leben pulsierte und sportliche Wettkämpfe stattfanden. Um die Vorstellungskraft anzuregen, wurden einige wenige Säulen wieder aufgerichtet. Dies stößt natürlich auf den erbitterten Widerstand der Archäologen, die den Status Quo möglichst unverändert erhalten wollen. Der Einsatz moderner Technik stellt eine Lösung dar.

Die Vision ist, dass der Besucher beim Betreten eine AR-Ausrüstung erhält entsprechen den Tonbandgeräten, die an einigen historischen Stätten schon jetzt verteilt werden. Diese Ausrüstung besteht aus einem Rechner und einer Datenbrille, in die zusätzliche Daten in das Sichtfeld eingeblendet werden können. An jeder beliebigen Stelle des Geländes kann der Besucher die Datenbrille aufsetzen und bekommt in das reale Bild der Umgebung eine virtuelle Rekonstruktion der alten Tempel eingeblendet. Im alten Stadium sieht er virtuelle Athleten, die historische Sportarten ausüben.

Abbildung A.3: Embassie Fahrsimulator als Anwendung für hoch dynamische Systeme

Zum Einblenden von virtuellen Gegenständen in eine reale Szene ist es erforderlich, die Position und Orientierung (Blickrichtung) des Anwenders auf dem Gelände genau zu bestimmen. Vom Rahmensystem wird ein hybrides Trackingsystem verwendet. In einem ersten Schritt wird eine grobe Bestimmung von Position und Orientierung mittels differential GPS und eines elektronischen Kompass durchgeführt. Wenn der Anwender Punkte auf dem Gelände erreicht, an denen Augmentierungen vorhanden sind, wird diese grobe Positionsbestimmung im zweiten Schritt durch ein Videotrackingsystem verfeinert.

A.3 Embassi-Fahrsimulator

Das Projekt Embassi-Fahrsimulator hat zum Ziel, neue User-Interface-Konzepte im Umgang mit technischen Geräten im Haushalt und im KFZ zu entwickeln (siehe Abbildung A.3). Dabei ist es insbesondere beim KFZ problematisch die sogenannte *Usability*, d.h. die Ergonomie dieser Geräte zu testen: im wirklichen Verkehrsgeschehen die *Usability* eines Abstandswarngerätes zu testen, ist zu gefährlich und zu teuer.

Hier ist der Einsatz von VR-Techniken sinnvoll. Dabei geht es primär nicht um die perfekte Simulation des Fahrverhaltens eines Fahrzeugs, als vielmehr um das

Nachstellen von typischen Verkehrssituationen zum Beispiel auf einer Autobahn.

Die Versuchsperson, die den Fahrsimulator benutzt, soll in etwa vom Verkehrsgeschehen so beansprucht werden wie bei einer realen Fahrt. Dabei wird beobachtet, wie die Versuchspersonen mit den technischen Geräten an Bord, zum Beispiel Autoradio und Navigationsgerät, zurechtkommt. Damit die technischen Geräte (zum Beispiel Abstandswarner) auch bezogen auf die dargestellte Verkehrssituation arbeiten können, simuliert der Fahrsimulator diverse Sensoren wie Tachometer, GPS, Kompass und Abstandssensoren und liefert simulierte Sensordaten an die im Testaufbau befindlichen Geräte.

Darüber hinaus protokolliert der Fahrsimulator diverse Parameter wie zum Beispiel Stellung von Lenkrad und Pedale, aktuelle Fahrspur, Abweichung von der Mitte der Fahrspur etc. Diese Daten erlauben es den an der Auswertung der Versuche beteiligten Ergonomen, Rückschlüsse auf die Belastung des Fahrers und den Grad der Ablenkung bei der Bedienung eines Geräts zu ziehen.

Die virtuellen Verkehrsteilnehmer im Simulator verhalten sich vollkommen realistisch, d.h. sie folgen dem Straßenverlauf, bremsen vor Hindernissen ab und überholen sich gegenseitig. Dieses Verhalten ist dabei keine vorprogrammierte Sequenz von Animationen. Vielmehr folgt jedes Fahrzeug einem Fuzzy-Logic-Regelsystem. Bei jedem Simulationsschritt wird für jedes Fahrzeug berechnet, wie weit es von der Mitte der Fahrspur abweicht, wie groß der Abstand zum Vordermann ist, ob die linke Fahrspur ausreichend Platz zum Ausscheren bietet oder ob die rechte Spur wieder frei zum Einscheren ist.

Basierend auf diesen Daten berechnet das Regelsystem für jedes virtuelle Fahrzeug Lenkradstellung und Beschleunigung. Diese Vorgehensweise erlaubt die Simulation eines äußerst realistischen, intelligenten Verhaltens der virtuellen Verkehrsteilnehmer.

A.4 Zusammenfassung

Die Beispiele zeigen die Bandbreite der Anwendungen und Anforderungen auf, die mit dem in dieser Arbeit vorgestellten Rahmensystem abgedeckt werden können. Applikationen sind keine monolithischen Blöcke, sondern entstehen durch das flexible Zusammenspiel von unterschiedlichen Komponenten.

Anhang B

Zusammenfassung und Ausblicke

Zusammenfassend werden die Ergebnisse der Arbeit vorgestellt und ein Ausblick als Anregung für weitere Entwicklungen und Forschungen aufgezeigt.

B.1 Zusammenfassung

Im Rahmen dieser Arbeit wird gezeigt, dass in traditionellen MR-Systemen vorwiegend Szenengraphen zur Organisation von strukturellen und graphischen Elementen zum Einsatz kommen. Diese MR-Systeme bieten zusätzlich eine Klassifikation von virtuellen Geräten und deren Abstraktion, um physikalische Ein- und Ausgabegeräte anzusprechen. Es wird weiterhin dargestellt, dass bisher zur Modellierung der verbindenden Applikationslogik und -dynamik nur ungenügende Modelle bereitstehen.

Aus diesem Grund wurde ein einheitliches Modell von Graphen entwickelt, das es erlaubt, alle dynamischen Aspekte einer MR-Applikation mit Hilfe von Komponenten und typisierten Kanten zu modellieren. Das Konzept abstrahiert die klassischen topologischen Beziehungen, sowie statische und dynamische Nachrichtenkanäle als Kanten innerhalb unterschiedlicher gerichteter Graphen. Daraus abgeleitete Anforderungen werden diskutiert und entsprechende Lösungen entwickelt. Dabei werden insbesondere Problemstellungen wie Erweiterbarkeit, die Verwaltung und Bereitstellung von statischen und dynamische Metabeschreibungen und die korrekte Verarbeitung von Nachrichten untersucht, und neuartige Ansätze entwickelt. Durch das einheitliche Modell wird der Entwicklungsprozess zur Erstellung von MR-Anwendungen gegenüber dem Stand der Technik wesentlich vereinfacht und beschleunigt.

Aufbauend auf einem einheitlichen Komponentenmodell werden neue Interaktionsmechanismen bereitgestellt. Dabei wird gezeigt, dass bisherige MR-Systeme sich meist auf eine statische Abstraktion von virtuellen Gerätegruppen beschränken und kaum darauf aufbauende Interaktionsmodelle anbieten. Aus diesem Grund wird in dieser Arbeit ein neuartiges, mehrstufiges Sensorkonzept entwickelt und vorgestellt, das unabhängig von Geräteklassifikationen Interaktionsverfahren beinhaltet, welche abhängig von der Ausprägung der Laufzeitumgebung interaktive Elemente bereitstellt. Es werden drei aufeinander aufbauende Sensorgruppen entwickelt, die als Komponenten der dynamischen und hierarchischen Graphen zum Einsatz kommen:

Die *Data Stream Sensor* (DSS) Objekte erlauben die direkte Anbindung von Ein- und Ausgabeströmen, unabhängig von einzelnen Geräten oder Interaktionsaufgaben. Sie sind nicht an Geräteklassen gebunden, sondern abstrahieren einzelne, typisierte Datenströme von oder zu Geräten.

Die *Direct Manipulation Sensor* (DMS) Objekte erlauben dem Anwender Teile der Szene graphisch-interaktiv und direkt zu manipulieren. Diese Sensoren reagieren auf Veränderungen des Benutzermodells, sind selbst aber unabhängig von konkreten Eingabegeräten.

Die *Indirect Manipulation Sensor* (IMS) Objekte ermöglichen der Anwendung Parameter vom Benutzer zu erfragen, und definieren somit konkrete Eingabeaufforderungen und Aufgaben. Sie beinhalten aber keine einzelnen graphischen Repräsentationen, sondern wählen selbständig, abhängig von der Laufzeitumgebung eine geeignete Interaktionsform aus. Das im Rahmen dieser Arbeit entwickelte Sensormodell stellt eine wesentliche Verbesserung gegenüber bestehenden Systemen dar. Es erlaubt die effiziente Entwicklung von Applikationen, die unabhängig von physikalischen und virtuellen Geräten und der eingesetzten Laufzeitumgebung sind.

Ein weiterer Schwerpunkt dieser Arbeit ist die Entwicklung von Verfahren zur Abbildung der Parallelverarbeitung von Prozessen und deren Skalierbarkeit. Es wird anhand von aktuellen Entwicklungen in der Prozessortechnologie gezeigt, dass es für die optimale Auslastung der Resourcen notwendig ist, möglichst viele Aufgaben und Verfahren zur Laufzeit auf unterschiedliche Prozesse aufzuteilen. Hierbei werden die typischen Abläufe in einem VR-System untersucht und eine Lösung entwickelt, die zwei unterschiedliche Ansätze der Parallelisierung beinhaltet. Einerseits wird die klassische Trennung von Applikation und Darstellung unterstützt und verallgemeinert, andererseits wird zusätzlich ein neuartiges Ver-

fahren entwickelt und vorgestellt, das die automatische Parallelisierung auf den Ereigniskanten der Applikationsgraphen bereitstellt. Der entscheidende Vorteil gegenüber den bestehenden Verfahren ist die selbständige und autarke Ermittlung von Teilgraphen, die parallel abgearbeitet werden können. Mithilfe dieser Lösung können Applikationen sehr effizient verteilt werden, aber dennoch sind die Ressourcen realer Maschinen begrenzt. Um globale Applikationsvorgaben erfüllen zu können, wie zum Beispiel das Darstellen mit einer fixen Bildwiederholrate oder die Animation einer variierenden Anzahl von Figuren pro Zyklus, ist eine globale Skalierbarkeit der Kosten gefordert. Zur Lösung dieses Problems wird ein globales Modell einwickelt und eingeführt, das automatisch lokale Kosten der Prozesse zur Darstellung und Veränderung skaliert. Dazu werden unterschiedliche skalierbare Verfahren untersucht und teilweise entscheidend verbessert. Das in dieser Arbeit entwickelte Modell zur Parallelisierung und Skalierung von MR-Systemen ist eine wesentliche Verbesserung gegenüber den bisherigen Verfahren, da es auf der Grundlage der globalen Zielvorgaben und dem Applikationsgraphen selbständig und automatisch den Resourcenverbrauch regelt und verteilt. Somit wird der Prozess der Anwendungsentwicklung wesentlich vereinfacht, da die aufwendige und fehlerträchtige manuelle Parallelisierung entfällt.

B.2 Ausblicke

Die hier vorgestellten Ergebnisse sind im Avalon Rahmensystem implementiert. Darüber hinaus gibt es Aspekte, die in dieser Arbeit nicht berücksichtigt werden konnten. Als Anregung und möglichen Gegenstand weiterer Entwicklungen und Forschungen werden sie im Folgenden kurz vorgestellt.

B.2.1 Integration von physikalischen Beziehungen

Das in Kapitel 3 vorgestellte Modell zur Modellierung von Beziehungen zwischen Komponenten erlaubt die Verwaltung von unterschiedlichen Beziehungstypen. Es stellt generische Mechanismen bereit, um diese Beziehungen zur Laufzeit auszuwerten. Konkret wurden Verfahren entwickelt, um Hierarchien von Objekten und explizite und implizite Nachrichtenkanten abzubilden. Weitere Kanten und entsprechende Manager zum Auswerten dieser Typen, wie zum Beispiel physikalische Beziehungen, sind nur teilweise umgesetzt und implementiert.

B.2.2 Parallelverarbeitung von zyklischen Graphen

Das in Kapitel 5 vorgestellte Konzept zur automatischen Parallelisierung von Aufgaben mit Hilfe der Ereigniskanten arbeitet auf separaten Teilgraphen um aufwendige Locking-Mechanismen zu vermeiden. Dies funktioniert sehr gut für typische VR-Applikationen, da im allgemeinen aufwendige Simulatoren und Animationen am Ende eines Logikbaums angesiedelt sind. Dennoch ist zu untersuchen, ob es möglich und sinnvoll ist, die Parallelisierung auf kleinere Einheiten auszudehnen. Dazu müssen die Veränderungen der Komponenten und die Verarbeitung von Nachrichten von unterschiedlichen Eingängen synchronisiert und abgesichert werden. Deshalb ist es notwendig Locking-Mechanismen einzuführen, um Dead-Lock Situation in zyklischen Graphen zu vermeiden.

B.2.3 Integration von weiteren skalierbaren Teilverfahren

In Kapitel 5 wurden nur einige skalierbare Verfahren zur Animation und Darstellung entwickelt und vorgestellt. Das System ist aber nicht auf diese konkreten Verfahren beschränkt, sondern offen für alle Methoden, die ihre Anwendungs- bzw. Darstellungskosten näherungsweise linear variieren können. In den letzten Jahren wurden in der Computergraphik eine Vielzahl von geeigneten Verfahren entwickelt, die zum Beispiel Punkt-Wolken, NURBS-Flächen oder Terrain-Modelle darstellen. Darüber hinaus gab es Arbeiten, die skalierbare Verfahren zur Animation und Simulation, zum Beispiel skalierbare Partikelsysteme, entwickelten und deren Integration durchaus sinnvoll sind.

Literaturverzeichnis

[1] Advanced Realtime Tracking GmbH. ARTtrack. http://www.ar-tracking.de/, 2002.

[2] K. Akeley. RealityEngine Graphics. In *SIGGRAPH*, pages 109–116. ACM, 1993.

[3] Alias Systems Corp. Alias/Maya Home page. http://www.alias.com, 2005.

[4] G. Altomare, O. Berlen, D. Rotondi, and A. Scopece. DICOM: An approach to the MHS UA design. In *Proc. IFIP WG 6.5: Message Handling Systems and Distributed ApplicationsO*, pages 385–396, Costa Mesa, California, October 10-12 1988. North-Holland.

[5] Inc. Apple Computer. *3D graphics programming with QuickDraw 3D*. Addison Wesley, 1995.

[6] Apple Computer Inc. *Apple Human Interface Guidelines*. Apple Computer Inc, 1 Infinite Loop, Cupertino, CA 95014, 2002.

[7] Ascension Technology Corporation. Bird Tracker, 1998.

[8] Autodesk Inc. Discreet/3DMax Home Page. http://www.discreet.com/, 2005.

[9] Michael Bajura, Henry Fuchs, and Ryutarou Ohbuchi. Merging virtual objects with the real world: Seeing ultrasound imagery within the patient. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 203–210, July 1992.

[10] Johannes Behr and Marc Alexa. Volume Visualization in VRML. In *Web3d - VRML 2001 Proceedings*, pages 23–27, 2001.

[11] Johannes Behr and Marc Alexa. Fast and Effective Striping. In *1. OpenSG Symposium, 2002, Darmstadt*, 2002.

[12] Johannes Behr, Patrick Daehne, and Marcus Roth. Utilizing X3D for Immersive Environments. *Web2D Symposium*, page 71, 2004.

[13] Johannes Behr, Torsten Froehlich, Christian Knoepfle, Bernd Lutz, Dirk Reiners, Frank Schoeffel, and Wolfram Kresse. The Digital Cathedral of Siena - Innovative Concepts for Interactive and Immersive Presentation of Cultural Heritage Sites. *ICCHIM Conference Proceedings, Milan*, 2001.

[14] Johannes Behr and Marc Niemann. Interactive Volume Data Rendering for Medical VR Applications. *Second Yomaa Symposium on Medical Imaging, Seoul, Korea*, 1998.

[15] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, C. Rebollo, and M. Fernandez. Multiresolution triangle strips, 2001.

[16] Calum Benson, Adam Elman, and Seth Nickell. GNOME Human Interface Guidelines. http://developer.gnome.org/projects/gup/hig/, 2002. The GNOME Usability Project.

[17] Allan Bierbaum, Albert Baker, Carolina Cruz-Neira, Patrick Hartling, Christopher Just, and Kevin Meinert. VR Juggler: A Virtual Platform for Virtual Reality Application Development. Master's thesis, Iowa State University, 2000.

[18] Allen Bierbaum. Clusterjuggler. In *VR2002, Open-Source-VR Coursenotes*, Orlando, March 2002.

[19] Roland Blach, Juergen Landauer, Angela Roesch, and Andreas Simon. A flexible prototyping tool for 3d realtime user-interaction. *Proceedings of 4th Eurographics workshop*, 1998.

[20] Roland Blach, Juergen Landauer, Angela Roesch, and Andreas Simon. A Highly Flexible Virtual Reality System. *Future Generation Computer Systems*, 14(3–4):167–178, 1998.

[21] Jasmin Blanchette and Jasmin Summerfield. *C++ GUI Programming With Qt*. Bruce Parens, 2004.

[22] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885ff, Nov 2000.

[23] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh – a generic and efficient polygon mesh data structure, 2002.

[24] David M. Bourg. *Physics for Game Developers, Enriching Game Content with Physics-based Realism*. O'Really, 2003. ISBN 5-384-334-3.

[25] Doug A. Bowman, Ernst Kruijff, Joseph Laviola, and Ivan Poupyrev. *3D User Interfaces, Theory and Practice*. Addison-Wesley, 2005. ISBN 0-201-75867-9.

[26] Grigore C. Burdea and Philippe Coiffet. *Virtual Reality Technology*. Wiley-Interscience, second edition, 2003. ISBN 0-471-36089-9.

[27] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *CHI*, pages 486–493, 2000.

[28] Matthew Conway, Randy Pausch, Rich Gossweiler, and Tommy Burnette. Alice: A rapid prototyping system for building virtual environments. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 2, pages 295–296, April 1994.

[29] Aldus Corporation and Microsoft Corporation. Tag image file format (TIFF) specification revision 5.0. Technical report, Aldus Corporation, 411 First Avenue South, Suite 200, Seattle, WA 98104, Tel: (206) 622-5500, and Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717, Tel: (206) 882-8080, August 8 1988.

[30] Carolina Cruz-Neira and Daniel J. Sandin. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *ACM Computer Graphics, SIGGRAPH 93*, 1993.

[31] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. *Proceedings of SIGGRAPH 93*, pages 135–142, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[32] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical Report TR93-027, Department of Computer Science, University of North Carolina - Chapel Hill, May 1 1994.

[33] Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, and Arie Kaufman. High-quality volume rendering using texture mapping hardware. *Eurographics Graphics Hardware Workshop*, pages 69–76, 1998.

[34] Patrick Dähne and Helmut Seibert. Managing Data Flow in Interactive MR Applications. *WSCG*, 2005.

[35] Stephan Diehl and Joerg Keller. Vrml with constraints. In *VRML '00: Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML)*, pages 81–86. ACM Press, 2000.

[36] ECMA International. ECMAScript: A general purpose, cross-platform programming language. ISO/IEC 16262:2002, 2002.

[37] Jihad A. El-Sana, Elvir Azanli, and Amitabh Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 131–138, San Francisco, 1999.

[38] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc., 2001.

[39] R. Eckert et al. Functional description of the graphical core system GKS as a step towards standardization. In E. Brauer, editor, *Informatik-Berichte*, volume 11, page 163. Springer Verlag, 1977.

[40] S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett. Virtual environment display system. In Frank Crow and Stephen M. Pizer, editors, *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pages 77–87, October 1986.

[41] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990. Overview of research to date.

[42] Torsten Fröhlich. *Dynamisches Objektverhalten in virtuellen Umgebungen*. PhD thesis, Technische Universität Darmstadt, Fachgebiet Graphisch-Interaktive Systeme, 2002.

[43] M. Garland. Multiresolution modeling: Survey & future opportunities. Technical report, Eurographics, STAR, 1999.

[44] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[45] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 263–270, 1998.

[46] Michael Garland and Yuan Zhou. Quadric-based Simplification in any Dimension. *ACM Transactions on Graphics*, 24(2), 2005.

[47] Ghee S. Programming Virtual Worlds. In *ACM Siggraph Course Notes*, 1997.

[48] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, volume ISBN 0-201-63451-1. Addison Wesley, Reading Massachusetts, 1996.

[49] Volker Haug and Jeanine Indest. Rs/6000 7044 model 270 technical overview and introduction, 2001.

[50] T. He and A. Kaufman. Virtual input devices for 3D systems. In *IEEE Visualization*, pages 142–148, 1993.

[51] Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[52] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[53] Hugues Hoppe. Efficient implementation of progressive meshes. *Computers and Graphics*, 22(1):27–36, 1998.

[54] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. *Computer Graphics*, 27(Annual Conference Series):19–26, 1993.

[55] Hugues H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 59–66, San Francisco, 1999.

[56] Hiroshi Hosobe. A geometric constraint library for 3d graphical applications. In *SMARTGRAPH '02: Proceedings of the 2nd international symposium on Smart graphics*, pages 94–101. ACM Press, 2002.

[57] T. Howard, R. Hubbold, and A. Murta. Maverik: A virtual reality system for research and teaching, 1999.

[58] Roger Hubbold, Jon Cook, Martin Keates, Simon Gibson, Toby Howard, Alan Murta, and Adrian West. Gnu/maverik a micro-kernel for large-scale virtual environments, 1999.

[59] Roger J. Hubbold, Xiao Dongbo, and Simon Gibson. MAVERIK - the manchester virtual environment interface kernal. In M. Göbel, J. David, P. Slavik, and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualization '96*, pages 11–20. Springer-Verlag Wien, 1996.

[60] INI-GraphicsNet. Avalon Projektseite. http://www.ini-graphics.net/avalon, 2001.

[61] InterSense. IS-900 Tracker. http://www.isense.com/, 2001.

[62] M. Isenburg and J. Snoeyink. Compressing polygon meshes as compressable ASCII. In *Proceedings of Web3D'02 Symposium*, pages 1–10, 2002.

[63] Martin Isenburg and Jack Snoeyink. Binary compression rates for ascii formats. In *Proceeding of the eighth international conference on 3D Web technology*, pages 173–ff. ACM Press, 2003.

[64] D. Jiang and J. P. Singh. Improving parallel shear-warp volume rendering on shared address space multiprocessors. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 252–263, 1997.

[65] Dirk Reiners Johannes Behr, Gerrit Voss. A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters. *rth Eurographics Workshop on Parallel Graphics and Visualisation 2002. Proceedings*, 2002.

[66] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors, 2001.

[67] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, July 1993.

[68] J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language*. 3Dlabs, Inc., 2004.

[69] Christian Knöpfle. *Intuitive und Immersive Interaktion für virtuelle Umgebungen*. PhD thesis, Fachbereich Informatik der Technischen Universität Darmstadt, 2003.

[70] K. Konstantinides and J.R. Rasure. The Khoros software development environment for image and signal processing. In *IEEE Transactions on Image Processing*, volume 3, pages 243–252, 2004. ISSN: 1057-7149.

[71] W. Kresse, D. Reiners, and C. Knöpfle. Color Consistency for Digital Multi-Projector Stereo Display Systems: The HEyeWall and The Digital CAVE. In *In Proceedings of IPT 2003, Zurich*, 2003.

[72] W. Krueger. The Application of Transport Theory to Visualization of 3D Scalar Data Fields. In *IEEE Visualization*, 1990.

[73] W. Krüger and B. Fröhlich. The Responsive Workbench. *IEEE Computer Graphics and Applications*, 1994.

[74] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors, 2003.

[75] l Russell, M. Taylor, T. Hudson, A. Seeger, H. Weber, J. rey, J. Aron, and T. Helser. Vrpn: a device-independent, network-transparent vr peripheral system, 2001.

[76] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(Annual Conference Series):451–458, 1994.

[77] Jesse Liberty. *Programming C#*. O'Reilly, 2003. 3rd Edition.

[78] Peter Lindstrom. Out-of-core simplification of large polygonal models. In Kurt Akeley, editor, *Siggraph 20000, Computer Graphics Proceedings*, pages 259–262. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[79] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *IEEE Visualization*, pages 279–286, 1998.

[80] Logitech. Spacemouse Interaction Device. http://www.spacemouse.com/, 1994.

[81] W.E. Lorensen and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21 (4), 1987.

[82] Juval Löwy. .Net Components. *O'Reilly*, 2003.

[83] Blair MacIntyre. A Turing Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. *ISWC*, 1997.

[84] Pattie Maes. *Concepts and Experiments in Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[85] T. H. Massie and J. K. Salisbury Salisbury. The phantom haptic interface: A device for probing virtual objects. In *ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 1994.

[86] Frank McPherson. *PocketPC*. Mc Graw Hill Osboarne, 2004. Second Edition.

[87] MediaMachines. FLUX Home-Page. http://www.mediamachines.com/, 2005.

[88] Stan Melax. A Simple, Fast and Effective Polygon Reduction Algorithm. *GameDeveloper*, 11, 1998.

[89] Microsoft. *Microsoft Windows User Experience, Official Guidelines for User Interface Developers and Designers*. Mircosoft Press, 1999.

[90] Microsoft. DirectDraw Surface (DDS) File Reference. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/DDSFileReference/ddsfileformat.asp, 2004.

[91] Sun Microsystems. Sun enterprise 450 features & benefits, 2001.

[92] Paul Milgram and Fumio Kishino. A Taxonomy of Mixed Reality Visual Displays. *IEICE Transactions on Information and Systems (Special Issue on Networked Reality)*, E77-D(12):1321–1329, 1994.

[93] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinitereality: A real-time graphics system. *Proceedings of SIGGRAPH 97*, pages 293–302, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[94] Gordon E. Moore. The complexity for minimum component cost. *Electronics Magazine*, 1965.

[95] J. Paul Morrison. *low-Based Programming: A New Approach to Application*. Van Nostrand Reinhold, 1994. ISBN 0442017715.

[96] Stephan Müller. Virtual Reality and Augmented Reality. In *International Converence on Visual Computing, Goa, India*, 1999.

[97] OpenGL Architectural Review Board. OpeGL Specification 2.0. http://www.opengl.org/documentation/spec.html, 2004.

[98] Hewlett Packard. Hp 9000-n specifications, 2001.

[99] Palm One Inc. Palm Handheld. http://www.palmone.com, 2004.

[100] Wayne Piekarski, Bruce Thomas, David Hepworth, Bernard Gunther, and Victor Demczuk. An architecture for outdoor wearable computers to support augmented reality and multimedia applications. *In Proc. of the rd International Conference on Knowledge-Based Intelligent Information Engineering System*, 2000.

[101] Polhemus Inc. FastTrack. http://www.polhemus.com/, 1992.

[102] I. Pourpyrev, S. Weghorst, M. Billinghurst, and T. Ichikawa. A Framework and Testbed for Studying Manipulation Techniques for Immersive VR. *ACM VRST*, pages 21–28, 1997.

[103] Bruno Raffin. Netjuggler. In *VR2002 Open-Source-VR Coursenotes*, Orlando, March 2002.

[104] Dirk Reiners. *OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications*. PhD thesis, Technische Universität Darmstadt, 2002.

[105] Dirk Reiners, Gerrit Voss, and Johannes Behr. Opensg: Basic concepts. In *1. OpenSG Symposium OpenSG 2002*, 2002.

[106] Brandon Reinhart. Mod Authoring for Unreal Tournament. http://www.unreal.epicgames.com/doc/moding, 1999.

[107] Gerhard Reitmayr and Dieter Schmalstieg. OpenTracker - An Open Software Architecture for Reconfigurable Tracking based on XML. In *IEEE Virtual Reality*, 2001.

[108] Patrick Reuter, Johannes Behr, and Marc Alexa. An improved adjacency data structure for efficient trianglestripping. *Journal of Graphics Tools*, 2005.

[109] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *iggraph/eurographics workshop on graphics hardware*, pages 109–118. ACM Press, 2000.

[110] Stefan Roettger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection, 2000.

[111] J. Rohlf and J. Helman. IRIS Performer: A high performance toolkit for real-time 3D graphics. *ACM Computer Graphics, SIGGRAPH 94*, 1994.

[112] Marcus Roth. IDEAL, entwicklung eines client-server systems zur handhabung multidimensionaler interaktionsgeräte in 3d echtzeit-graphiksystemen. Master's thesis, Fachhochschule Mannheim, 1998.

[113] Marcus Roth. Integration paralleler rendering-verfahren für lose gekoppelte systeme mit opensg. In *OpenSG 2002 Workshop*, 2002.

[114] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[115] J. Van Scheltinga, J. Smit, and M. Bosma. Design of an On-Chip Reflectance Map. In *Eurographics Workshop on Graphics Hardware*, pages 51–55. Eurographics, 1995.

[116] B. Schwald and P. Figueiredo. Learning of Rigid Point-Based Marker Models for Tracking with Stereo Camera Systems. In S. Müller, editor, *Virtuelle und Erweiterte Realität, 1. Workshop der GI-Fachgruppe VR/AR*, pages 23–34, Chemnitz, 2004.

[117] B. Schwald and C. Malerzcyk. Controlling Virtual Worlds Using Interaction Spheres. In C.A. Vidal, editor, *5th Symposium on Virtual Reality (SVR)*, Fortaleza, Brazil, 2002. Brazilian Computer Society.

[118] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc., 1999.

[119] Michael Shafae and Renato Pajarola. Dstrips: Dynamic triangle strips for real-time mesh simplification and rendering. In Jon Rokne, Wenping Wang, and Reinhard Klein, editors, *Proceedings Pacific Graphics 2003*, pages 271–280. IEEE, 2003.

[120] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. In *ACM Transactions on Information Systems*, volume 11, pages 287–317, 1993.

[121] William R. Sherman and Alan B. Craig. *Understanding Virtual Reality; Interface, Application, and Design*. Morgen Kaufmann, 2003. ISBN 1-55860-353-0.

[122] P. Shirley and A. A. Tuchman. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24, pages 63–70, 1990.

[123] J. Springer, H. Tramberend, and B. Fröhlich. On Scripting in Distributed Virtual Environments. In *4. Immersive Projection Technology Workshop*, 2000.

[124] R. Stiles, S. Tewari, and M. Mehta. Adapting VRML For Free-form Immersed Manipulation. *VRML 97, 3. Symposium on the Virtual Reality Modeling language*, 1998.

[125] R. Stiles, S. Tewari, and M. Metha. Adapting VRML 2.0 for Immersive Use. *VRML 97, Second Symposium on the Virtual Reality Modeling Language*, 1997.

[126] Matthias Stiller. Methoden zur Reduktion polygonaler Daten. Master's thesis, Fachhochschule Darmstadt, 1999.

[127] P.S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *ACM Computer Graphics*, 1992.

[128] Sun Microsystems Inc. *Java Look and Feel Design Guidelines*. Addison-Wesley Professional, 2001. ISBN: 0201775824.

[129] Teim Sweeney. Unreal Networking Architecture. http://www.unreal.epicgames.com/unreal/paper/networking, 1999.

[130] Tim Sweeney. UnrealScript Language References. *http://www.unreal.epicgames.com/unreal/paper/scripts*, 2000.

[131] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. ACM Press, 1998. ISBN 0-201-17888-5.

[132] H. Tago, K. Hashimoto, N. Ikumi, M. Nagamatsu, M. Suzuoki, and Y. Yamamoto. CPU for PlayStation II. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings, Munich, Germany*, pages 696–704, 2001.

[133] The Internet Engineering Task Force. The ZeroConf Network Protocoll Spezification. www.zeroconf.org, 2004.

[134] H. Tramberend. Avocado – a distributed virtual environment framework. In *Virtual Reality 1999. Houston, Texas*, pages 14–21, 1999.

[135] Henrik Tramberend. *Avocado: A Distributed Virtual Environment Framework*. PhD thesis, Technische Fakultät, Universität Bielefeld, 2003.

[136] Matthias Unbescheiden. *Physikalisch basierte Simulation in Virtuellen Umgebungen*. PhD thesis, Technische Universität Darmstadt, 1999.

[137] Allen VanGelder and Kwansik Kim. Direct Volume rendering with shading via three-dimensional textures. Technical Report UCSC-CRL-96-16, Symposium on Volume Visualization, 1996.

[138] Inc Virtock Technologies. Vizx3D Modelling System. http://www.vizx3d.com/, 2005.

[139] VPL research. DataGlove input device. www.vpl-research.com, 1988.

[140] W3C. Xml Protocol Working Group, sOAP Version 1.2 Specification. *http://www.w3.org/2000/xp/Group/*, 2000.

[141] Daniel Wagner and Dieter Schmalstieg. First Steps Towards Handheld Augmented Reality. In *Proceedings of the 7th International Conference on Wearable Computers, White Plains, NY, USA*, 2003.

[142] Peter Walsh. *The Zen of Direct3D Game Programming*. Prima Tech, 2002.

[143] Web3D Consortium. X3D framework & SAI, ISO/IEC 19775:200x. ISO, 2004. http://www.web3d.org/x3d/specifications/x3d_specification.html.

[144] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering via 3d textures. In *IEEE Symposium on Volume Visualization 2000*, 2000.

[145] Manfred Weiler and Thomas Ertl. Ein Volume-Rendering-Framework für OpenSG. In *1. OpenSG Symposium*, 2002.

[146] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *ACM SIGGRAPH 1998*, 1998.

[147] Paul Woodward. Powerwall. http://www.lcse.umn.edu/research/pw.html, 2001.

[148] Rolf Ziegler. *System zur integrieten Einsatz von haptischen Displays in virtuellen Umgebungen*. PhD thesis, Technische Universität Darmstadt, Fachgebiet Graphisch-Interaktive Systeme, 1998.

[149] Davis Ziff. Nvidia geforce6 / NV40. http://reviews.zdnet.co.uk/review/14/1/158.html, 2004.

Lebenslauf

Name:	Johannes Rudolf Behr	
Geburtsdatum/–ort:	21. Dezember 1968, Hassfurt in Bayern	
Staatsangehörigkeit:	deutsch	
Familienstand:	ledig	
Wohnort:	Taunusstrasse 25, 60329 Frankfurt	
Ausbildung:	1975 bis 1981	Volksschule Hofheim
	1981 bis 1985	Realschule Hofheim
	1986 bis 1988	Staatliche Fachoberschule Schweinfurt Abschluss Fachabitur
	1988 bis 1993	Studium der Infomatik an der Fachhochschule Würzburg Vordiplom 1989, Abschluss als Dipl. Inform. (FH) 1993
	1994 bis 1996	Master of Science Studiengang an der University of Wolverhampton Abschluss als MSc 1996
Anstellung:	1997 bis 2004	Wissenschaftlicher Mitarbeiter am Zentrum für Graphische Datenverarbeitung, Darmstadt
	2004 bis 2006	Wissenschaftlicher Mitarbeiter an der Technischen Universität Darmstadt

Veröffentlichungen

- Marcelo G. Malheiros, Johannes Behr, and Jorge Diz. An Extensible Interactive Image Synthesis Environment. echnical report DCA-006/97 - DCA, FEEC, Unicamp, 1997.
- Wu Shin Ting and Johannes Behr. An Extensible Interactive Image Synthesis Environment. XXIV Semish proceedings, 1997.
- Johannes Behr and Andreas Froehlich. Avalon, an Open VRML VR/AR system for Dynamic Application. Topics, 1(1):28, 1998.
- Johannes Behr and Marc Niemann. Interactive Volume Data Rendering for Medical VR Applications. Second Yomaa Symposium on Medical Imaging, Seoul, Korea, 1998.
- Axel Hildebrand Johannes Behr. Sanare - VR Med enviroment. Topics, 1998.
- Johannes Behr and Marcelo. Cooperative VR enviroment. ISBGA, 2000.
- Marc Alexa, Johannes Behr, and Wolfgang Mueller. The morph node. In Web3d - VRML 2000 Proceedings, pages 29-34. ACM Press, 2000. ISBN 1-58113-211-5.
- Behr J, Choi SM, GroÃkopf S, MH, and Sakas G. Modelling, visualization, and interaction techniques for diagnosis and treatment planning in cardiology. Computers & Graphics, Vol 24.5:741-753, 2000. ISSN 0097-8493.
- Stefan Grosskopf, Johannes Behr, Choi Soo-Mi. 3D Modellierung zur Diagnose und Behandlungsplanung in der Kardiologie. Der Radiologe, 40(3):256-261, 2000.
- Johannes Behr and Marc Alexa. Volume Visualization in VRML. In Web3d - VRML 2001 Proceedings, pages 23-27, 2001.
- Johannes Behr, Torsten Froehlich, Christian Knoepfle, Bernd Lutz, Dirk Reiners, Frank Schoeffel, and Wolfram Kresse. The Digital Cathedral of Siena - Innovative Concepts for Interactive and Immersive Presentation of Cultural Heritage Sites. ICCHIM Conference Proceedings, Milan, 2001.

- M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. IEEE Visualization 2001, pages 21-28, October 2001. ISBN 0-7803-7200-3.
- Torsten Froehlich Johannes Behr, Peter Eschler. Cybernarium Days 2002 - A Public Experience of Virtual and Augmented Worlds. First International Symposium on Cyber Worlds 2002, 2002.
- Dirk Reiners Johannes Behr, Gerrit Voss. A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters. rth Eurographics Workshop on Parallel Graphics and Visualisation 2002. Proceedings, 2002.
- Johannes Behr and Marc Alexa. Fast and Effective Striping. In 1. OpenSG Symposium, 2002, Darmstadt, 2002.
- Marc Alexa and Johannes Behr. Linear Geometry Interpolation in OpenSG. In 1. OpenSG Symposium OpenSG, Darmstadt, 2002.
- M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Computing and rendering point set surfaces, 2002.
- Dirk Reiners, Gerrit Voss, and Johannes Behr. Opensg: Basic concepts. In 1. OpenSG Symposium OpenSG 2002, 2002.
- Patrick Daehne Johannes Behr. Avalon: Ein komponentenorientiertes Rahmensystem fuer dynamische Mixed-Reality Anwendungen. TUD thema Forschung, 2003.
- Johannes Behr, Patrick Daehne, and Marcus Roth. Utilizing X3D for Immersive Environments. Web2D Symposium, page 71, 2004.
- Johannes Behr, Christian Knoepfle, and Daehne Patrick. A scalable sensor approach for immersive and desktop VR Application. In HCI/VR International, 2005.
- Patrick Reuter, Johannes Behr, and Marc Alexa. An improved adjacency data structure for efficient trianglestripping. Journal of Graphics Tools, Volume 4, 2005.